White Paper

# CENTRAL AUTHENTICATION SERVICE (CAS) SSO FOR EMC® DOCUMENTUM® REST SERVICES


## Abstract

This white paper provides a detailed review of Central Authentication Service (CAS) SSO integration with EMC Documentum REST Services by exploring the architecture and consumption workflow, the deployment recommendations and alternatives, and the troubleshooting for this integration.



January, 2014


EMC²

Copyright © 2014 EMC Corporation. All Rights Reserved.

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

The information in this publication is provided “as is.” EMC Corporation makes no representations or warranties of any kind with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a particular purpose.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on EMC.com.

Part Number h12766

## Table of Contents


Executive summary ..... 5
Audience ..... 5
Terminology ..... 5
Part I. CAS Feature Overview ..... 7
CAS Architecture ..... 7
Basic Authentication Flow ..... 7
Proxy Authentication ..... 10
CAS Extensions and Support ..... 13
CAS Integration with Documentum REST Services ..... 13
Architecture Overview ..... 13
Browser CAS SSO Walkthrough ..... 14
Non-browser CAS SSO Walkthrough ..... 17
CAS Proxy Negotiation ..... 20
Using Client Token ..... 21
Managing Timeout for Tokens ..... 23
Name ..... 23
Producer ..... 23
Timeout Policy ..... 23
Default ..... 23
Single Sign-out ..... 25
Future Possibilities ..... 26
Part II. CAS Configuration and Troubleshooting ..... 27
CAS Server Configuration ..... 27
Preliminaries ..... 27
Obtaining CAS Server Binary ..... 27
Configuring CAS RESTful API ..... 29
Configuring CAS Properties ..... 29
Configuring LDAP mapping ..... 30
Configuring Service Registry ..... 32
Customizing CAS Proxy Response ..... 33
Validating CAS Deployment ..... 33
Content Server Configuration ..... 35
Preliminaries ..... 35
Enabling CAS SSO Plugin ..... 38
Validating Content Server Configuration ..... 39

REST Server Configuration ..... 40
Preliminaries ..... 40
Enabling CAS Authentication Scheme ..... 40
Validating REST Deployment ..... 41
Advanced CAS Integration Options ..... 45
Multiple Authentication Schemes ..... 45
CAS SSO across Content Server Repositories ..... 48
Security Configuration for Client Token ..... 49
REST Server Clustering for CAS ..... 50
CAS Server Clustering ..... 53
Logging and Troubleshooting ..... 57
Content Server Logging ..... 57
REST Server Logging ..... 59
CAS Server Logging ..... 60
REST Server Troubleshooting ..... 61
Content Server Troubleshooting ..... 62
CAS Server Troubleshooting ..... 63
Client Side Troubleshooting ..... 64
Conclusion ..... 70
References ..... 70

Table 1 Terminology ..... 5
Table 2 Expiration Policy of Client Token ..... 22
Table 3 Full View of Token Timeout ..... 23

Figure 2 CAS Proxy Authentication Flow ..... 10
Figure 3 Browser Client CAS SSO for Documentum REST Services ..... 14
Figure 4 Non-browser Client CAS SSO for Documentum REST Services ..... 18
Figure 5 Proxy Negotiation between REST Server and CAS Server ..... 20
Figure 7 Basic and CAS Authentication Work Together ..... 45
Figure 8 Setup Sub Group for PGT Storage ..... 52

# Executive summary

Central Authentication Service (CAS) is an enterprise Single Sign-On (SSO) solution for web services. SSO means a better user experience when running a multitude of web services, each with its own means of authentication. With a SSO solution, different web services may authenticate to one authorized source of trust that the user needs to log in to, instead of requiring the end-user to log in into each separate service. This white paper provides a complete overview of CAS SSO for the EMC Documentum REST Services 7.1 release, which includes the following:

- Architecture and the authentication flow
- Deployment details, including both basic and advanced environment setup
- Troubleshooting recommendations
- Samples

## Audience

This white paper is intended for architects, engineers, support professionals and customers. It provides the information needed for enabling CAS SSO for Documentum REST Services.

## Terminology

Special terms, abbreviations and acronyms that may appear in this guide are defined below:

### Table 1 Terminology

| Term | Description |
|---|---|
| Single Sign-on (SSO) | A property of access control of multiple related, but independent software systems that a user logs in once and gains access to all systems without being prompted to log in again at each of them. |
| Central Authentication Service (CAS) | An open SSO protocol for web access. In most cases, it also<br>refers to the open-source Java server that provides enterprise Single Sign-On solution for web services. |
| CAS Service | A CAS SSO enabled web application that clients try to access. |

| | |
|---|---|
| CAS Proxy | A CAS service that accesses other CAS services on behalf of a particular user. |
| CAS Target | A service that accepts proxied credentials from at least one particular proxy. |
| Ticket Granting Ticket (TGT) | A ticket produced by CAS and held by the user client as an authenticated identity. |
| CAS Ticket Granting Ticket Cookie (CASTGC) | A TGT carried in the format of a cookie. |
| Service Ticket (ST) | A ticket produced by CAS<br>Clients send STs to services. Each ST is used only once for access to one specific service. |
| Proxy Granting Ticket (PGT) | A ticket produced by CAS and held by a proxy to confer the ability to produce proxy tickets. |
| Proxy Granting Ticket IOU (PGTIOU) | A ticket sent by CAS alone in a service validation response, and with a PGT to the callback URL. |
| Proxy Ticket (PT) | A ticket usable by a proxy to access a target by impersonating a single user. |
| Client Token (CT) | A token produced by Documentum REST Server and carried as a cookie for an authenticated CAS identity. The user client can use the CT to access REST resources without going through the CAS SSO again. |
| Login Ticket (LT) | A ticket produced by Content Server to act as a temporary password for an authenticated user. |
| Secure Sockets Layer (SSL) | SSL is a cryptographic protocol which is designed to provide communication security over the Internet. |
| Jasig | A consortium of educational institutions and commercial affiliates sponsoring open source software projects for higher education, including the CAS implementation. The website is: http://www.jasig.org/ |
| Ehcache | An open source, standards-based cache for boosting performance, offloading your database, and simplifying scalability. The website is: http://ehcache.org/ |

# Part I. CAS Feature Overview

## CAS Architecture

CAS is a single sign-on protocol for the web. Its purpose is to permit a user to access multiple applications while providing their credentials (such as user id and password) only once. The CAS protocol involves at least three parties:

- a client web browser
- the web application requesting authentication (called CAS Service)
- the CAS server

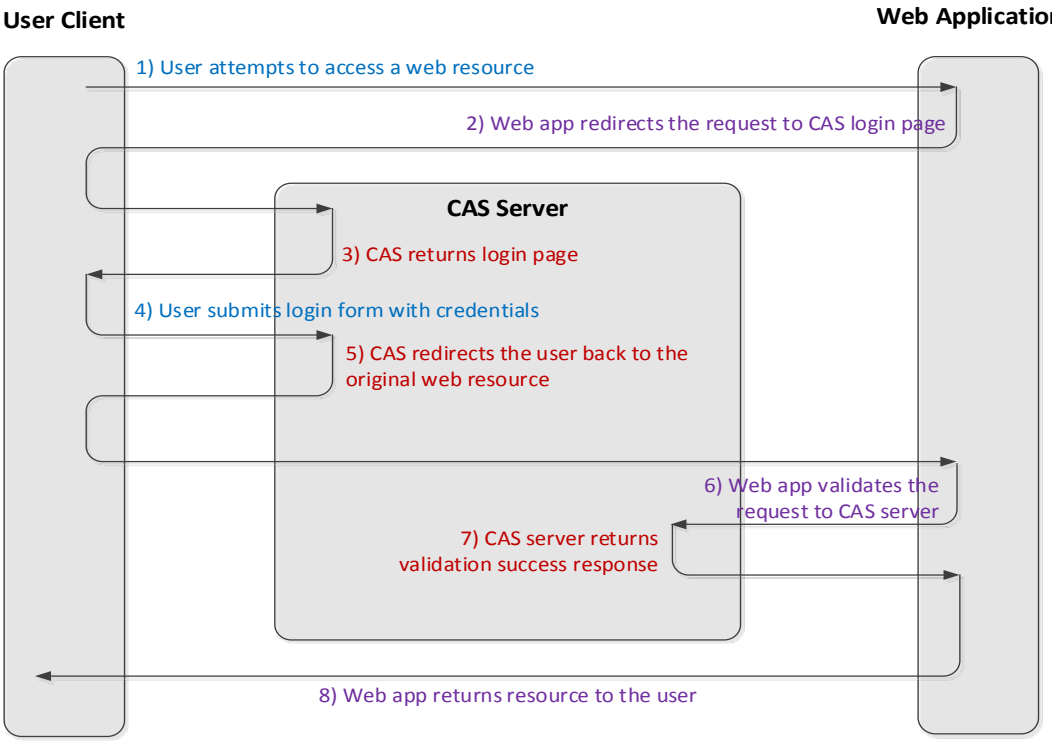
It may also involve a back-end service (called CAS Target), such as a database server, that does not have its own HTTP interface but communicates with a web application.

There are several open source CAS server implementations for the CAS protocol. The Jasig CAS server is the most widely adopted by the industry. In Documentum REST Services, the CAS server we talk about is referring to the Jasig CAS Server implementation. The Jasig CAS is designed as a standalone Java web application (http://www.jasig.org/cas ). There are also a number of CAS clients developed to facilitate web applications to complete the CAS authentication.

### Basic Authentication Flow

In this section, we go through the basic authentication flow of the CAS SSO. From the flow, we will get the basic idea of the CAS SSO.

Figure 1 Basic CAS Authentication Flow


User Client
Web Application
1) User attempts to access a web resource
2) Web app redirects the request to CAS login page
CAS Server
3) CAS returns login page
4) User submits login form with credentials
5) CAS redirects the user back to the original web resource
6) Web app validates the request to CAS server
7) CAS server returns validation success response
8) Web app returns resource to the user

### Step 1):

The end-user A client requests the URI of a web application. The web application leverages a CAS server to provide the authentication service. Under these assumptions, the client comes to access a specific web resource on the web server, e.g. http://web-app/about.me

### Step 2):

If the web application checks the request and finds no CAS tickets along with the request, it will redirect the request to a CAS login URI. The request URI is typically validated by a CAS client agent (e.g. a Spring Security filter) which is deployed in the same context as the web application. The CAS client agent redirects the end-user request to the CAS server login. This redirection also occurs when a CAS ticket for the request is not valid. Please see Step 5 to know what a valid request URI looks like.

The redirecting URI must contain a '*service*' parameter whose value is the original web resource URI. This is important for below two reasons:

- The CAS server next will produce a ST for the resource access. Since each ST is specific to a unique resource and one time use, the CAS server must know which service URI the user is attempting to access.
- The CAS server needs to know the original service URI to redirect the user back after a successful authentication.

A redirecting back URI follows this pattern: *http://cas-server/login?service=http://web-app/about.me*

**Step 3):**

The CAS server returns an HTML page which contains a login form.

**Step 4):**

The user client posts the user id and password to the CAS server.

**Step 5):**

Next, the CAS server must validate the user credential against a back-end user directory server, which is usually an identity provider or LDAP server. The CAS server must also check whether the service URI within '*service'* parameter (e.g. *http://web-app/about.me*) has been registered as a CAS Service. If not, the authentication fails even if the user credential is correct.

After validation, the CAS server redirects the user client back to the original service URI. In addition to that, CAS appends the information below in the redirecting response:

- An ST is appended to the original service URI which represents the authorized access to the web resource and will be later validated by the web application. So the redirecting back URI is like: *http://web-app/about.me?ticket=ST1bdgbwHlReBonmaudvxJlcas*

- A cookie called '*CASTGC'* is set back to the user client which is essentially a TGT.

The ST is used only once but the TGT can be reused to represent one authenticated user. When the user client attempts to access another web resource, the CASTGC cookie is reused to request a new ST without needing to provide the user credential again, as long as the CASTGC is not expired. The TGT is held by the user client and remains private between the user client and CAS server. It will not be sent to the web application.

**Step 6):**

The web application validates the request. Specifically, it obtains the '*ticket'* parameter from the request URI and validates it to the CAS server. Same to Step 2), this is usually done by a CAS client agent in the web application.

**Step 7):**

The CAS Server returns the validation success response. The CAS server may additionally return some user attributes after the validation so that the web

application can make access control for the web resources based on the user attributes.

**Step 8):**

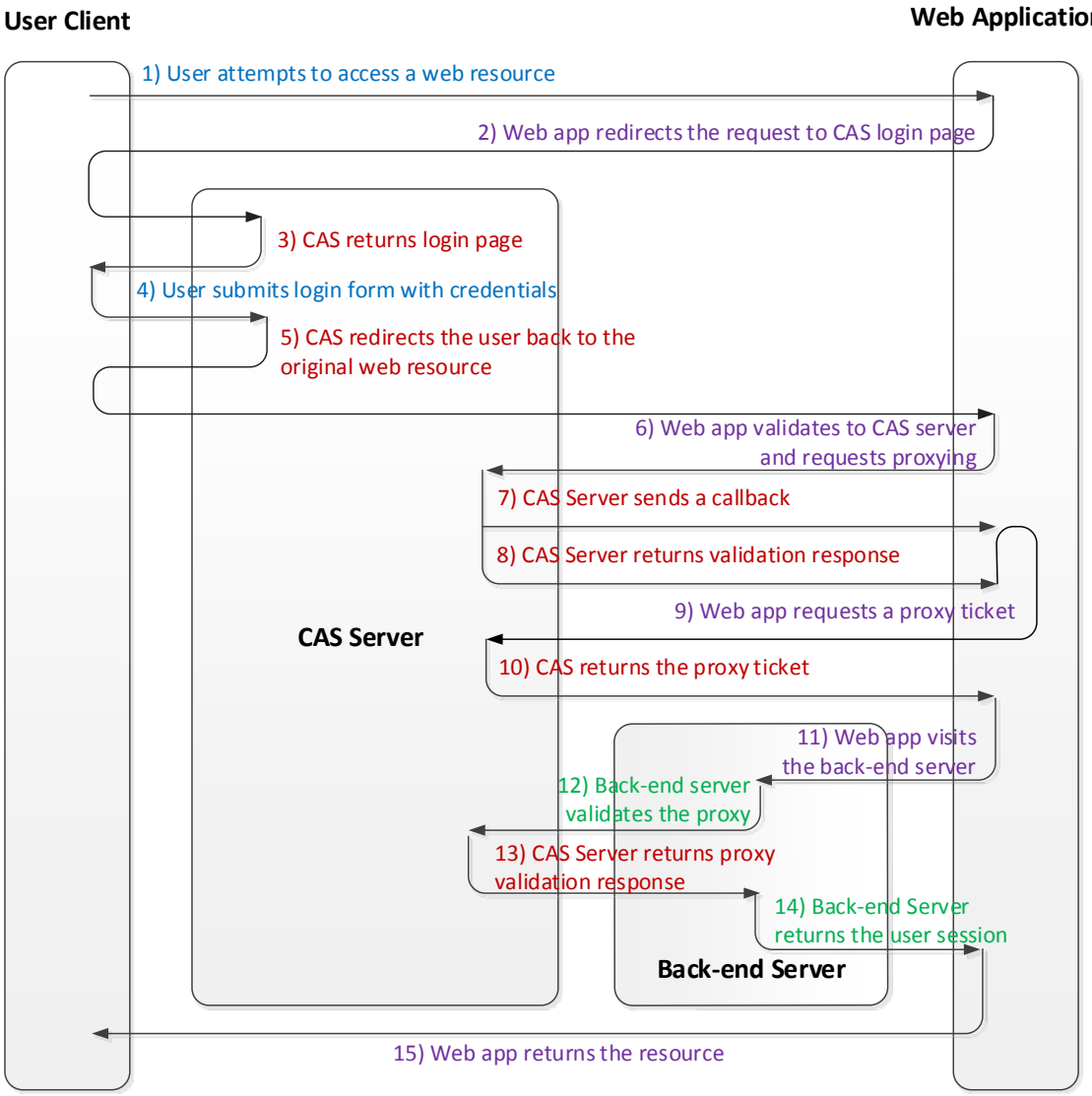
Web application returns the web resource to the user client.

### Proxy Authentication

As Documentum REST Services is a multi-layer system, it uses CAS proxy authentication to achieve Single Sign-on.

In a multi-tier CAS installation, an agent acting on behalf of a user, but without direct access to the user's cookie cache, may need to convince a third party that it represents the user legitimately. So in CAS 2.0, proxy authentication is introduced to support the authentication delegation in a multi-layer system. Its authentication flow is presented in the diagram below.

Figure 2 CAS Proxy Authentication Flow


User Client
Web Application
1) User attempts to access a web resource
2) Web app redirects the request to CAS login page
3) CAS returns login page
4) User submits login form with credentials
5) CAS redirects the user back to the original web resource
6) Web app validates to CAS server and requests proxying
7) CAS Server sends a callback
8) CAS Server returns validation response
9) Web app requests a proxy ticket
CAS Server
10) CAS returns the proxy ticket
11) Web app visits the back-end server
12) Back-end server validates the proxy
13) CAS Server returns proxy validation response
14) Back-end Server returns the user session
Back-end Server
15) Web app returns the resource

Step 1) to Step 5) are the same as basic authentication flow. Starting with Step 6), the flow is as follows.

### Step 6):

The requested resource is on the back-end server instead of the web application. Therefore, the web application must to request a PT to access the back-end server to get the resource. In this step, the web application sends the ST validation request to the CAS server and it appends a PGT callback URL to the request, asking for proxy authentication after the ST validation.

To perform proxy authentication, the web application first needs to get a PGT (just like TGT), then use the PGT to request a PT for a specific resource from the back-end server. During the ST validation, the CAS Server sends a PGT to the web application.

**Step 7):**

After the ST validation, the CAS server will further enable proxying for the web application. The PGT is sent to the web application in two steps. In Step 7), the CAS server sends a PGTIOU-PGT mapping to the callback URI which is specified by the web application. The web application needs to have storage to save the PGTIOU-PGT mapping.

**Step 8):**

After the callback succeeds, the CAS server then responds to the validation request by Step 6). In the validation response, the CAS server tells the web application a PGTIOU token.

The web application is responsible to go to the PGTIOU-PGT mapping storage to lookup the PGT based on the input of the PGTIOU.

**Step 9):**

The web application requests a PT for the Back-end Server access using the obtained PGT.

**Step 10):**

The CAS server validates the PGT and returns a PT to the web application. Similarly as the web application, the back-end server must be registered as a CAS service to enable the proxied authentication.

**Step 11):**

The web application visits the back-end server with the PT.

**Step 12):**

The back-end server validates the PT to the CAS server.

**Step 13):**

The CAS server returns the proxy validation responses to the back-end server.

**Step 14):**

The back-end server returns the user session to the web application.

**Step 15):**

The web application then uses the session to operate the object and returns the web resource to the user client.

The CAS proxy authentication is adopted by Documentum REST Services for its CAS SSO. There are many required steps to complete the CAS SSO, thus, in Documentum REST Services we have introduced an optimized approach which enables the users to follow the CAS authentication workflow only for once and then use an authenticated token to access resource subsequently. Please refer to *CAS Integration with Documentum REST Services* for details.

### CAS Extensions and Support

The CAS SSO provides an authentication framework, and it can be integrated with other authentication schemes like LDAP authentication, Kerberos SSO, SAML SSO and so on. Therefore, once CAS has been integrated to a web application, it can do more than simple centralized authentication. In Documentum REST Services, we provide the basic capability of CAS SSO for the REST Services. Moreover, customers also have the option to integrate the CAS SSO with other authentication schemes. Please refer to Jasig community for extending the authentication schemes for your system.

In the CAS deployment required by Documentum REST Services, several CAS server extensions are required to be installed:

- CAS server integration with Ehcache
- CAS server integration with Restlet
- CAS server support for generic
- CAS server support for LDAP

All these extensions are open-source software and can be downloaded from the open Maven repository. The details will be illustrated in the section “

*Obtaining CAS Server* Binary”.

There are a number of ways to find support for CAS authentication, please refer to http://www.jasig.org/jasig-support for help.

## CAS Integration with Documentum REST Services

### Architecture Overview

Documentum REST Services supports CAS authentication to achieve a robust authentication and SSO infrastructure for both browser and non-browser clients. There are at least four parties taking part in the CAS SSO:

- A user agent, e.g. web browser, generic HTTP client, etc.
- a CAS server 3.5.2

- a Documentum REST Services server 7.1+
- a Content Server 7.1+

Documentum REST Services acts as a CAS proxy in the CAS SSO in that it needs to access another CAS service, Content Server, on behalf of the end user. Therefore, Content Server in the CAS SSO acts as a CAS target which accepts the proxied credentials.

Optionally, there could be other components involved in the SSO, such as an LDAP server, a load balancer, and a reverse proxy server for business specific deployment consideration.

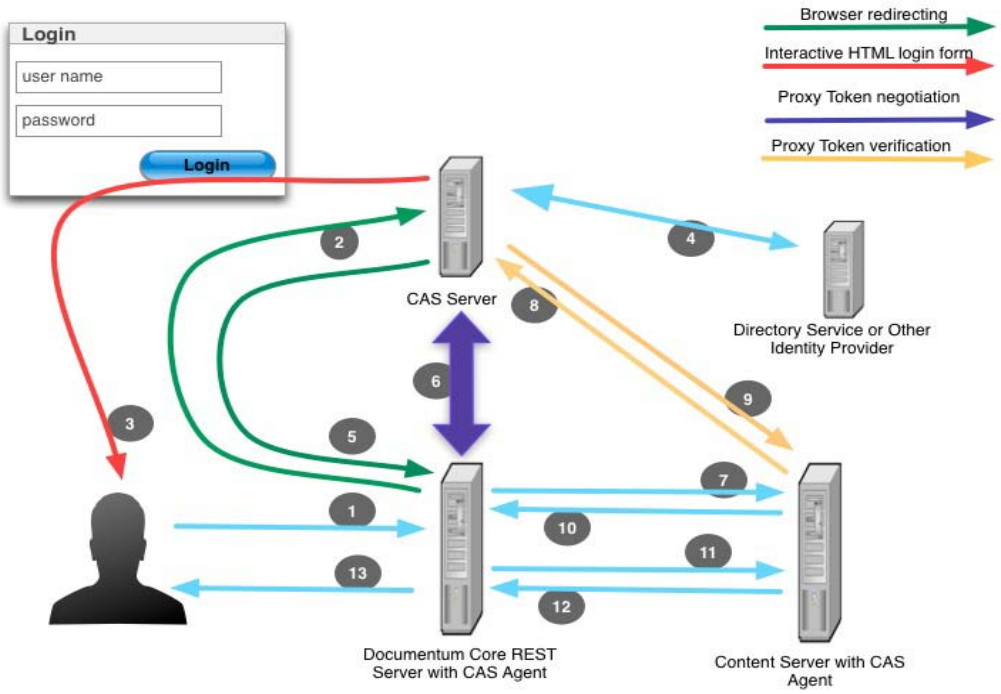
To achieve the practical CAS SSO performance, Documentum REST Services introduces an authentication token called Client Token (CT) to provide the pass-through access for CAS authenticated clients. This token is sent by the REST server to the client after initial CAS SSO authentication flow has completed successfully either after initial login or after token was renegotiated due to an expiry time. It is carried in the form of a 'DOCUMENTUM-CLIENT-TOKEN' cookie.

The CAS authentication supports browser and non-browser clients. The next sections elaborate on the authentication workflows for both scenarios.

### Browser CAS SSO Walkthrough

The following diagram illustrates the workflow of CAS authentication for a browser client. The browser CAS SSO is following the standard CAS proxy authentication.

Figure 3 Browser Client CAS SSO for Documentum REST Services




**Step 1):**

A browser client visits a Documentum Core REST Service resource, for instance, *http://restserver:8080/dctm-rest/repositories/acme01*

**Step 2):**

The REST server sends back a 302 redirecting response, asking the user to authenticate to the CAS server, for instance,

```
Response Status Code: 302 Moved Temporarily
Location: https://casserver:8443/cas/login?service= http%3A%2F%2Frestserver%3A8080%2Femc-rest%2Frepositories%2Facme01
```

**Step 3):**

The client inputs username/password and submits the request to the CAS Server

**Step 4):**

The CAS server connects to the directory service to verify the user credential

**Step 5):**

The CAS server returns back a 302 redirecting response, providing the ST and TGT for user to visit the resource, for instance

```
Response Status Code: 302 Moved Temporarily
Location: http://restserver:8080/emc-rest/repositories/acme01 ?ticket=ST-238-
mHMdsK0A9sAhie2T1dep-cas01.example.org
Set-Cookie: CASPRIVACY=""; Expires=Thu, 01-Jan-1970 00:00:10 GMT; Path=/cas/ CASTGC=TGT-165-
zm6GUY4gFJP3AjtY4EQiXJ7M5lIGJDiIevY6AZTlkOhg2F9J2B-cas01.example.org; Path=/cas/; HttpOnly;
Secure
```

**Step 6):**

The REST server negotiates a PT from the CAS server. Please refer to section "CAS Proxy Negotiation " for details.

**Step 7):**

The REST server calls Content Server by passing the PT.

**Step 8):**

Content Server's new CAS Plugin calls the CAS server to validate PT, for instance

```
https://casserver/cas/proxyValidate ?service=http://restserver:8080/emc-rest/repositories/acme01
&ticket=PT-957-ZuucXqTZ1YcJw81T3dxf
```

**Step 9):**

The CAS server validates PT and if succeeds, responds with the following message:

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
  <cas:authenticationSuccess>
    <cas:attribute name="dmCSLdapUserDN" value="CN=halbertj,OU=testou,DC=iigplat,DC=com"/>
    <cas:user>halbertj</cas:user>
    <cas:proxies>
      <cas:proxy>https://restserver:8443/pgtCallback</cas:proxy>
    </cas:proxies>
  </cas:authenticationSuccess>
</cas:serviceResponse>
```

Note that there is a custom attribute '*dmCSLdapUserDN*' returned in the validation response. This is a customized behavior of CAS deployment required by Documentum REST Services. This attribute is used by Content Server to validate the user distinguished name.

**Step 10):**

Content Server creates a session for user "*halbertj*" and generates a ticket the client will be able to use for subsequent calls.

**Step 11):**

The REST server sends actual operations to Content Server using the established session.

**Step 12):**

Content Server returns operation results to the REST server.

**Step 13):**

The REST server returns results to the browser client, including a DOCUMENTUM-CLIENT-TOKEN cookie for future resource access, for instance

```
Response Status Code: 200 OK
Content-Type: application/json;charset=UTF-8
Set-Cookie: DOCUMENTUM-CLIENT-TOKEN="Ym9ibGVlOkRNX1RJQ0tFVD1UMEpL...=="; Version=1;
Path=/emc-rest
{ "id": 15, "name": "acme01",.....
```

Later on, the browser client visits other resources using the cookie, for instance

```
GET http://restserver:8080/emc-rest/repositories/acme01/cabinets
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Cookie: DOCUMENTUM-CLIENT-TOKEN="Ym9ibGVlOkRNX1RJQ0tFVD1UMEpL...=="
```

The REST server calls Content Server passing the token credentials, then returns the result as resource response, for instance

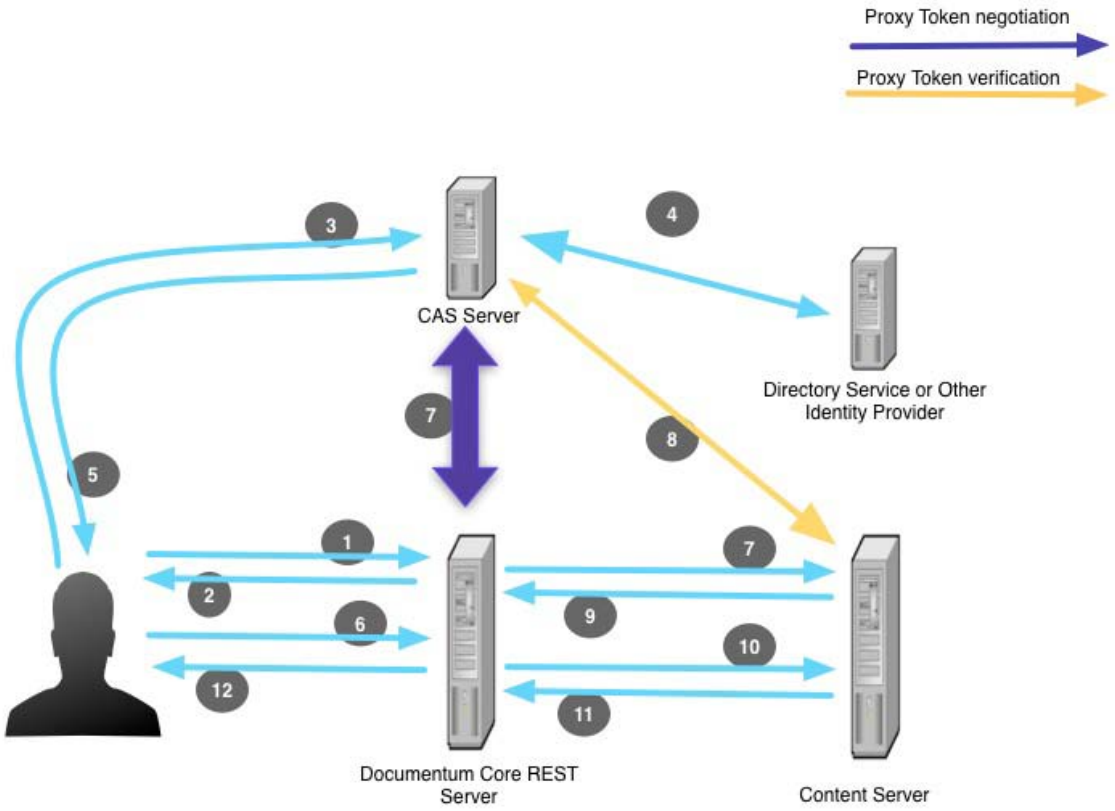
```
Response Status Code: 200 OK
Content-Type: application/json;charset=UTF-8

{ "offset": 0, "limit": 50, ...
```

## Non-browser CAS SSO Walkthrough

The redirecting mechanism does not make much sense in many scenarios. For instance, non-browser clients may not have the built-in ability to actually handle URL redirects; the redirected CAS login URL is pointed to an HTML form page which is aimed for human being interaction; and some browser clients (i.e. widgets in mash-ups) may never reach the intended caller with a 302 HTTP code. For all above scenarios, the alternative approach that authenticating clients to CAS with a pure RESTful API is more promising. We leverage the CAS RESTful API to achieve the CAS SSO for non-browser client. The introduction for CAS RESTful API can be found on Jasig wiki https://wiki.jasig.org/display/CASUM/RESTful+API. It requires the CAS deployment to include extension '*cas-server-integration-restlet*'. Here is the authentication flow for non-browser CAS SSO.

Figure 4 Non-browser Client CAS SSO for Documentum REST Services




**Step 1):**

An non-browser client visits a Documentum REST resource, for instance, *http://restserver:8080/dctm-rest/repositories/acme01*

**Step 2):**

The REST server sends back a response based on the client redirecting preference. If the client request has set an HTTP header ***DOCUMENTUM-NO-CAS-REDIRECT=true***, REST server returns 401 with the CAS RESTful login URL.

```
========== Request ===============
GET http://restserver:8080/emc-rest/repositories/acme01
DOCUMENTUM-NO-CAS-REDIRECT: TRUE
Host: restserver:8080
 ========== Response ===============
Status code: 401 Unauthorized
Location: https://casserver:8443/cas/v1/tickets
WWW-Authenticate : CAS realm="my realm"
```

Otherwise, REST server just returns 302 for redirecting, same to the browser client flow.

**Step 3):**

The non-browser client gets the login URI for CAS RESTful API from the response Location header, and posts to CAS server to get TGT, for instance

```
POST https://casserver:8443/cas/v1/tickets
Host: casserver:8443

username=halbertj&password=breakme
```

**Step 4):**

The CAS server validates the user against the directory service or other identity provider.

**Step 5):**

The CAS server returns back a TGT resource location, for instance

```
Status code: 201 Created
Location: https://casserver:8443/cas/v1/tickets/TGT-166-
AHw7Sv5wFnVtWaUQZzxOTRc5YxiGnMJPEVWyai0mFeTccjwnWa-cas01.example.org
```

With this location, the non-browser client posts to CAS server to get ST for the specific resource URI, for instance,

```
========== Request ===============
POST https://casserver:8443/cas/v1/tickets/TGT-166-
AHw7Sv5wFnVtWaUQZzxOTRc5YxiGnMJPEVWyai0mFeTccjwnWa-cas01.example.org
Host: casserver:8443
service=http%3A%2F%2Frestserver%3A8080%2Femc-rest%2Frepositories%2Facme01
========== Response ===============
Status code: 200 OK
ST-239-rYiYHoQJ2ZhJopdMuxjl-cas01.example.org
```

Note that URL encoding is required for the service URL in the request body. CAS server returns the ST directly in its response.

**Step 6):**

The non-browser client posts to the REST server to consume the resource, with ST appended as query parameter, for instance,

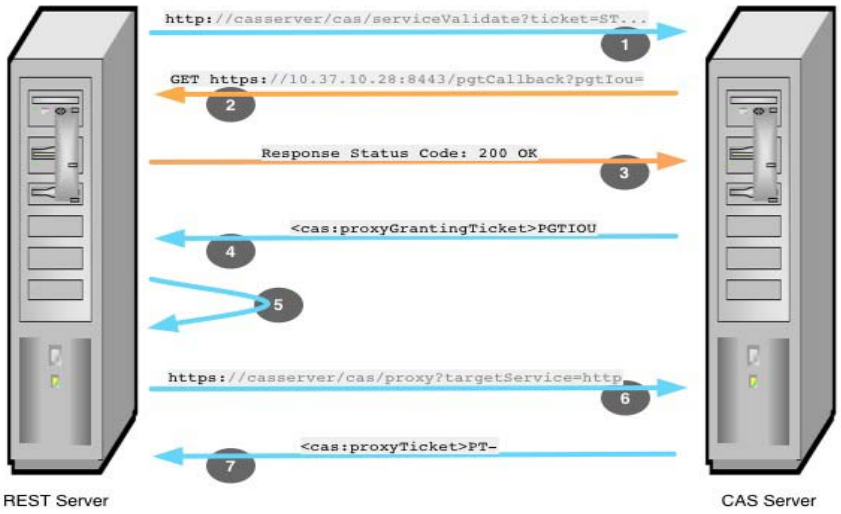
```
GET http://restserver:8080/emc-rest/repositories/acme01?ticket=ST-239-rYiYHoQJ2ZhJopdMuxjl-
cas01.example.org
Host: restserver:8080
```

The rest of the flow is exactly the same as browser client flow, seeing browser client flow from Steps 6) to Step 13).

## CAS Proxy Negotiation

This section describes Step) 6 in browser SSO flow, and for the same, Step) 7 in non-browser SSO flow. As mentioned, in the CAS SSO, the authentication from the REST server to Content Server is using CAS proxy. The CAS proxy requires callback to complete the PGT negotiation. The diagram below illustrates how a PGT is obtained during the ST validation. This happens between the REST server and CAS server and is totally opaque to the user client.

### Figure 5 Proxy Negotiation between REST Server and CAS Server




### Step 1):

The REST server calls the CAS server to validate the ST sent by the user client, and it asks to callback the PGT, for instance

```
GET http://casserver/cas/serviceValidate?ticket=ST-238-mHMdsK0A9sAhie2T1dep-
cas01.example.org&service=http://restserver:8080/emc-
rest/repositories/acme01&pgtUrl=https://restserver:8443/pgtCallback
Accept text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

### Step 2):

Prior to responding the validation request, the CAS server makes a callback to "*pgtUrl*" passing PGTIOU and PGT in the URI parameters.

```
GET https://restserver:8443/pgtCallback?pgtIou=PGTIOU-85-
8PFx8qipjkWYDbuBbNJ1roVu4yeb9WJIRdngg7fzl523Eti2td&pgtId=PGT-330-
CSdUc5fCBz3g8KDDiSgO5osXfLMj9sRDAI0xDLg7jPn8gZaDqS
```

**Step 3):**

The callback service responds to the CAS server with OK status. Typically, the callback service is implemented within the REST server and it saves the PGTIOU and PGT mapping in a local storage.

**Step 4):**

The CAS server responds to the initial ST validation request and additionally sends back a PGTIOU token in the response.

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
  <cas:authenticationSuccess>
    <cas:user>halbej</cas:user>
    <cas:proxyGrantingTicket>PGTIOU-85-
8PFx8qipjkWYDbuBbNJ1roVu4yeb9WJIRdngg7fzl523Eti2td</cas:proxyGrantingTicket>
  </cas:authenticationSuccess>
</cas:serviceResponse>
```

**Step 5):**

The REST server looks for the exact PGT from callback service storage using the PGTIOU as the key.

**Step 6):**

The REST server calls the CAS server to get a PT for Content Server access by passing the PGT.

```
GET https://casserver/cas/proxy ?targetService=ContentServer&pgt=PGT-330-
CSdUc5fCBz3g8KDDiSgO5osXfLMj9sRDAI0xDLg7jPn8gZaDqS
```

**Step 7):**

The CAS server responds with a PT. With the PT, the REST server is able to login to Content Server.

## Using Client Token

The purpose of producing a CT is to provide an authenticated REST client with a temporary and expirable token to access the REST server without a need to negotiate a new session ticket. The CT cookie has no session state stored on the REST server. Therefore, it works in cluster environment as well. It contains an encrypted Documentum login ticket (LT) and additional metadata used for further pass-through

authentication. A CT cookie is encrypted and validated by the REST server. The REST client is not expected in any means to persist or decrypt the token. A validation failure of the CT cookie leads to CAS authentication failure.

The CT is expirable and we expose two options in *rest-api-runtime.properties* to set the timeout for a CT cookie.

- *rest.security.client.token.timeout*

It's the expiration for the CT in seconds. The default is 3600.

- *rest.security.client.token.expiration.policy*

The use of the policy is explained in below table.

Table 2 Expiration Policy of Client Token

| Policy | Description |
|---|---|
| *com.emc.documentum.rest.security.ticket.impl.HardTimeoutExpirationPolicy* | The client token expires after a specified duration.<br><br>If the REST client sends a request before the duration, the REST server accepts the client token. If the REST client sends a request after the duration, the REST server rejects the client token, and the client has to authenticate again. |
| *com.emc.documentum.rest.security.ticket.impl.TolerantTimeoutExpirationPolicy* (default) | The client token expires after two times of the specified duration. The REST server issues new client tokens under certain conditions. For details, see the following:<br><br>• If the REST client sends a request before the duration, the REST server accepts the client token.<br>• If the REST client sends a request after the duration and before two times of the duration, the REST server accepts the client token and issues another client token with the same duration to the client for subsequent requests.<br>• If the REST client sends a request after two times of the duration comes to an end, the REST server rejects the client token and the client has to authenticate again. |
| *com.emc.documentum.rest.security.ticket.impl.TouchedTimeoutExpiratio* | The client token expires after a specified duration. The REST server issues new client tokens under certain conditions. For details, see the following: |

| | |
|---|---|
| *nPolicy* | • If the REST client sends a request before the duration, the REST server accepts the client token, and issues another client token with the same duration to the client for subsequent requests.<br>• If the REST client sends a request after the duration, the REST server rejects the client token, and the client has to authenticate again. |

The CT cookie is encrypted with cryptography algorithms. Users can choose different crypto options for the CT cookie during the REST service deployment. For instance, the security provider other than RSA, the crypto algorithm other than AES, the key size other than 128, etc. Please refer to section "*Security Configuration for Client Token*" for details.

The CT cookie is by default used for a single REST server. Therefore, the CT cookie produced from one REST server cannot be used by another REST server. To make the CT work across multiple REST servers, please refer to section "*REST Server Clustering for CAS*" for details.

The CT cookie is by default used by single repository access. Therefore, the CT cookie produced based on one repository login cannot be used to access REST resources in the other repository. To make the CT work across repositories, please refer to section "*CAS SSO across Content Server Repositories*".

### Managing Timeout for Tokens

In the full CAS authentication flow, there are multiple tokens and cookies produced by the authorities. Here is the table showing the tokens and their timeout policies in the CAS authentication.

#### Table 3 Full View of Token Timeout

| Name | Producer | Involved Parties | Timeout Policy | Default |
|---|---|---|---|---|
| TGT cookie | CAS Server | • CAS Server<br>• REST Client | • TimeoutExpirationPolicy<br>• HardTimeoutExpirationPolicy | TimeoutExpirationPolicy (7200 seconds) |

| | | | | |
|---|---|---|---|---|
| | | | • ThrottledUseAndTimeoutExpirationPolicy<br>• NeverExpiresExpirationPolicy | |
| ST | CAS Server | • CAS Server<br>• REST Server<br>• REST Client | • MultiTimeUseOrTimeoutExpirationPolicy | NeverExpiresExpirationPolicy (10 seconds) |
| PGT | CAS Server | • CAS Server<br>• REST Server | Same to TGT | Same to TGT |
| PT | CAS Server | • CAS Server<br>• REST Server<br>• Content Server | Same to ST | Same to ST |
| CT cookie | REST Server | • REST Server<br>• REST Client | • HardTimeoutExpirationPolicy<br>• TouchedTimeoutExpirationPolicy<br>• TolerantTimeoutExpirationPolicy | TouchedTimeoutExpirationPolicy (3600 seconds) |
| LT | Content Server | • Content Server<br>• REST Server<br>• REST Client | Internally controlled by CT | Internally controlled by CT |

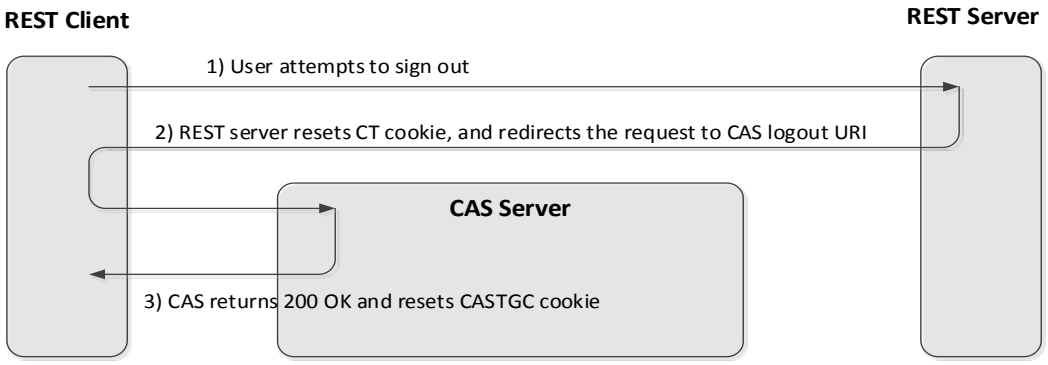
Upon the CAS authentication flow, ST and PT tokens are used for only once. It is assumed that the user client (or the REST server regarding to PT) sends the ST token for validation immediately after obtaining the token. As a result, the ST token has very short lifetime. The TGT token is held by the user client for a relatively long period to request multiple STs upon on-demand requests, so it has much longer living time. The PGT token is used internally by the REST server to request PTs, so it does not affect the end user interactions. The CT cookie is held by the user client to access on-demand REST resources, so it has a relatively longer living time, too. The LT token's validness is controlled by the CT token, so it has no impact to the end user.

For the robust security configuration, the timeout for all these tokens and cookies must be managed carefully. Typically, the timeout of ST should be less than 30 seconds; the timeout of TGT should be more than 15 minutes which prevents the web application from returning the session timed-out error to the end user frequently; the timeout for the CT should be between the ST and the TGT, usually from 10 minutes to one hour. The timeout of CT needs to be shorter than the time of TGT because when the end user gets the timed-out error using an expired CT cookie to access some resources, he/she still has the chance to use the valid TGT to negotiate new CT cookies without needing to send the private credentials over the wire again. For a browser client, this is usually done silently with the cookie handling and URL redirecting mechanisms supported by the web browser.

## Single Sign-out

CAS authentication utilizes CT cookies to simplify the communication between an authenticated REST client and the REST server. Moreover, CAS authentication supports Single Sign-out that invalidates both the CT cookie and the CAS TGT cookie. When a client explicitly logs out, the session is terminated and the client has to negotiate a new session ticket. The following diagram shows the flow for CAS Single Sign-out.

### Figure 6 CAS Single Sign-out Flow


REST Client
REST Server
1) User attempts to sign out
2) REST server resets CT cookie, and redirects the request to CAS logout URI
CAS Server
3) CAS returns 200 OK and resets CASTGC cookie

The following workflow explains the single sign-out process in more detail:

1) A client sends a request to the REST server for logout by providing a CT.

```
GET https://rest-server:8443/cas/logout
Cookie : DOCUMENTUM-CLIENT-TOKEN=AYQEVn....DKrdst
```

2) The REST server validates the CT and resets it, and then redirects the REST client to the CAS server for logout.

```
HTTP/1.1 302 302 Moved Temporarily
Set-Cookie: DOCUMENTUM-CLIENT-TOKEN=""; Expires=Thu, 01-Jan-1970 00:00:10 GMT; Path=/dctm-rest/
Location: https://cas-server:8443/cas/logout
```

The REST client resets the client side CASTGC cookie and access the CAS server for logout.

```
GET https://cas-server/cas/logout
Cookie : CASTGC= TGT-AYQEVn....DKrdst
```

3) The CAS server destroys the TGT from its memory entry and sends back HTTP 200 with an empty CASTGC cookie.

```
HTTP/1.1 200 OK
Set-Cookie: CASTGC= ""; Expires=Thu, 01-Jan-1970 00:00:10 GMT; Path=/cas/
```

5) The REST client resets its client side TGT cookie, and the single sign-out finishes. Both TGT and CT are invalidated.

*Please note that the actual Documentum CT token cannot be invalidated by the server side. It is just be cleared up on the client side. The actual invalidation of the CT token depends on its own expiry. For strict security requirement, it is recommended to set a short default time out for the CT token.*

## Future Possibilities

CAS SSO not only provides the basic authentication for LDAP integration, but also allows extending its authentication protocol to work with other authentication protocols. The Jasig CAS space https://wiki.jasig.org/display/CASUM/Home has a list of possible authentication mechanisms that can be integrated with CAS. For example, CAS server can work as a delegation party for Kerberos protocol that it negotiates the Kerberos/SPNEGO ticket between the end user and the Kerberos KDC. What it means is that if an IT system has already adopted some SSO security, it isn't necessary to replace the existing SSO with CAS for the usage of Documentum REST Services; instead, CAS can be integrated into the existing security infrastructure to achieve the enterprise SSO. CAS can be even integrated into federation security to meet the requirement of cloud enablement. We will explore the possibilities of the CAS extensions for Documentum REST Services in separate documentation.

# Part II. CAS Configuration and Troubleshooting

## CAS Server Configuration

### Preliminaries

Documentum REST Services 7.1 supports CAS Server version 3.5.2.

The CAS server is deployed in a Java web container as a WAR file. It must be installed within a web container such as Tomcat, JBoss, etc.

The ticket validation of CAS requires HTTPS, so it is required to enable SSL for the web container as well. For the formal product implementation, you need to register the SSL certificate to an authority. For the quick startup, create a certificate for your web container with any of the following tools. For instance, use Java keytool (http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html), Openssl (http://www.openssl.org/), or any other certificate tool (like Portecle, http://portecle.sourceforge.net/) to produce a certificate for the web container. After the certificate generation, import it to the JRE keystore of the CAS server with following command.

```
keytool -import -keystore <path-of-the-jre-cacert> -storepass "<password>" -alias "<cas-alias>" -file
<path-of-the-cas-certificate-file>
```

In addition to that, this certificate must be trusted by the REST server for proxy callback (mandatory in HTTPS), so **go to the REST server** and import it to the REST server's JRE trust store.

```
keytool -import -keystore <path-of-the-jre-cacert> -storepass "<password>" -alias "<cas-alias>" -file
<path-of-the-cas-certificate-file>
```

### Obtaining CAS Server Binary

The CAS server is an open-source project It can be download from Jasig source repository. However, Jasig doesn't publish the CAS server binary directly; instead, it allows downloading the Java source code for CAS server and lets the users to build it by themselves.

**Step 1):**

Go to download center of Jasig CAS project http://www.jasig.org/cas/download, and download CAS Server 3.5.2 Release.

**Step 2):**

Prior to building, please install Java 6 and Maven 2 in your local machine. Go to their websites to download and install them.

- Java http://www.oracle.com/technetwork/java/javase/downloads/index.html
- Maven http://maven.apache.org/

After installation, set both Java and Maven bin path into the system environment variable.

**Step 3):**

The CAS deployment for Documentum REST Services requires some additional extensions to be installed in the CAS server. So before building the CAS WAR file, unzip the downloaded CAS in local file system and add the following elements within the *‹dependencies›* element of the root *pom.xml*.

```
<dependency>
  <groupId>org.jasig.cas</groupId>
  <artifactId>cas-server-support-ldap</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.jasig.cas</groupId>
  <artifactId>cas-server-support-generic</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.jasig.cas</groupId>
  <artifactId>cas-server-integration-restlet</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>org.jasig.cas</groupId>
  <artifactId>cas-server-integration-ehcache</artifactId>
  <version>${project.version}</version>
</dependency>
```

**Step 4):**

After building the WAR file, the next step is to configure and build the CAS server. The precedence can be changed. In this sample, we will configure the CAS server after building. The CAS server can also be configured firstly prior to the build using the technology of MAVEN2 WAR overlay. For the latter, please refer to: https://wiki.jasig.org/display/CASUM/Best+Practice+-+Setting+Up+CAS+Locally+using+the+Maven+WAR+Overlay+Method.

**Step 5):**

Run the command "*mvn clean install –DskipTests=true*". And the *cas-server-webapp-3.5.2.war* file is generated under the following directory:

*\cas-server-3.5.2\cas-server-webapp\target*

### Configuring CAS RESTful API

The CAS RESTful API is used by non-browser clients. Update the file *cas-server-webapp-3.5.2.war/WEB-INF/web.xml* by adding the following elements:

```
<servlet>
  <servlet-name>restlet</servlet-name>
  <servlet-class>com.noelios.restlet.ext.spring.RestletFrameworkServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>restlet</servlet-name>
  <url-pattern>/v1/*</url-pattern>
</servlet-mapping>
```

### Configuring CAS Properties

The *cas.properties* file is located in: *cas-server-webapp-3.5.2.war/WEB-INF/*. This file contains the basic settings for the CAS server. Update '*server.name*' and '*host.name*' according to the corresponding instructions.

```
server.name=https://RSCAS02:8443
host.name=RSCAS02.EMC.COM
```

### Configuring LDAP mapping

The CAS LDAP mapping is configured at: cas-server-webapp-3.5.2.war/WEB-INF/deployerConfigContext.xml.

Add the following element in the bean of *credentialsToPrincipalResolvers*. This bean resolves CAS credentials to LDAP user attributes during login attempts.

```
<property name="credentialsToPrincipalResolvers">
  <list>
    <!-- other resolves beans here -->
    <!-- Documentum customization begin -->
    <bean class="org.jasig.cas.authentication.principal.CredentialsToLDAPAttributePrincipalResolver">
      <property name="credentialsToPrincipalResolver">
        <bean
class="org.jasig.cas.authentication.principal.UsernamePasswordCredentialsToPrincipalResolver"/>
      </property>
      <property name="filter" value="(sAMAccountName=%u)"/>
      <property name="principalAttributeName" value="sAMAccountName"/>
      <property name="searchBase" value="CN=Users,DC=ACME,DC=COM"/>
      <property name="contextSource" ref="contextSource"/>
      <property name="attributeRepository" ref="attributeRepository"/>
  </bean>
  <!-- Documentum customization end -->
</property>
```

Please note that the values highlighted in red are **specific to Microsoft Active Directory**. For other LDAP servers, the values could be different. The same assumption applies to other samples in this section.

Add the following element in the bean of *authenticationHandlers*. This bean resolves the LDAP user binding for CAS authenticating users.

```
<property name="authenticationHandlers">
  <list>
    <!-- other authentication handler beans here -->
    <!-- Documentum customization begin -->
    <bean class="org.jasig.cas.adaptors.ldap.BindLdapAuthenticationHandler"
              p:filter="sAMAccountName=%u"
      p:searchBase="CN=Users,DC=ACME,DC=COM"
      p:contextSource-ref="contextSource"
      p:ignorePartialResultException="true" />
  </list>
  <!-- Documentum customization end -->
</property>
```

Add the following element in the root. This bean configures the connection from the CAS server to the LDAP server.

```
<beans>
  <!-- other beans here -->
  <!-- Documentum customization begin -->
  <bean id="contextSource" class="org.springframework.ldap.core.support.LdapContextSource">
    <property name="pooled" value="false"/>
    <property name="url" value="ldap://192.168.0.8:389" />
    <property name="userDn" value="ACME\Administrator"/>
    <property name="password" value="Password123"/>
    <property name="baseEnvironmentProperties">
      <map>
        <entry key="java.naming.security.authentication" value="simple" />
      </map>
    </property>
  </bean>
  <!-- Documentum customization end -->
</beans>
```

Add the following element in the root. This bean resolves user attributes for authenticated users.

```
<beans>
  <!-- other beans here -->
  <!-- Documentum customization begin -->
  <bean id="attributeRepository"
class="org.jasig.services.persondir.support.ldap.LdapPersonAttributeDao">
    <property name="contextSource" ref="contextSource" />
    <property name="baseDN" value="CN=Users,DC=ACME,DC=COM" />
    <property name="requireAllQueryAttributes" value="true" />
    <property name="queryAttributeMapping">
      <map>
        <entry key="username" value="sAMAccountName" />
      </map>
    </property>
    <property name="resultAttributeMapping">
      <map>
        <entry value="dmCSLdapUserDN" key="distinguishedName"/>
      </map>
    </property>
</bean>
  <!-- Documentum customization end -->
</beans>
```

## Configuring Service Registry

The CAS LDAP mapping is configured at: *cas-server-webapp-3.5.2.war/WEB-INF/deployerConfigContext.xml*. The service registry holds the services that CAS server grants access permission. We need to register services for both Documentum REST Services and Content Server. There are basically three ways to configure the service registry.

- By beans – set service registry in Java beans in deployerConfigContext.xml
- By Service Management – set service registry in CAS service management UI
- By database – persist service registry to a database

We will show the sample of bean configuration. Open *deployerCOnfigContext.xml* and add the following element in the root:

```
<beans>
  <!-- other beans here -->
  <!-- Documentum customization begin -->
  <bean
  id="serviceRegistryDao"
  class="org.jasig.cas.services.InMemoryServiceRegistryDaoImpl">
  <property name="registeredServices">
    <list>
      <!-- the following registered service is for any HTTP protocol including REST API, as an example -->
      <bean class="org.jasig.cas.services.RegexRegisteredService">
        <property name="id" value="0" />
        <property name="name" value="HTTP" />
        <property name="description" value="Allows HTTP(S)protocols" />
        <property name="serviceId" value="^(https?)://.*" />
        <property name="evaluationOrder" value="10000001" />
      </bean>
      <!-- the following registered service is for Content Server, as an example -->
      <bean class="org.jasig.cas.services.RegexRegisteredService">
        <property name="id" value="1" />
        <property name="name" value="Content Server proxy service" />
        <property name="description" value="Allows Content Server service" />
        <property name="serviceId" value="ContentServer" />
        <property name="evaluationOrder" value="1" />
        <property name="allowedAttributes">
          <list><value>dmCSLdapUserDN</value></list>
        </property>
      </bean>
    </list>
  </property>
  <!-- Documentum customization end -->
</beans>
```

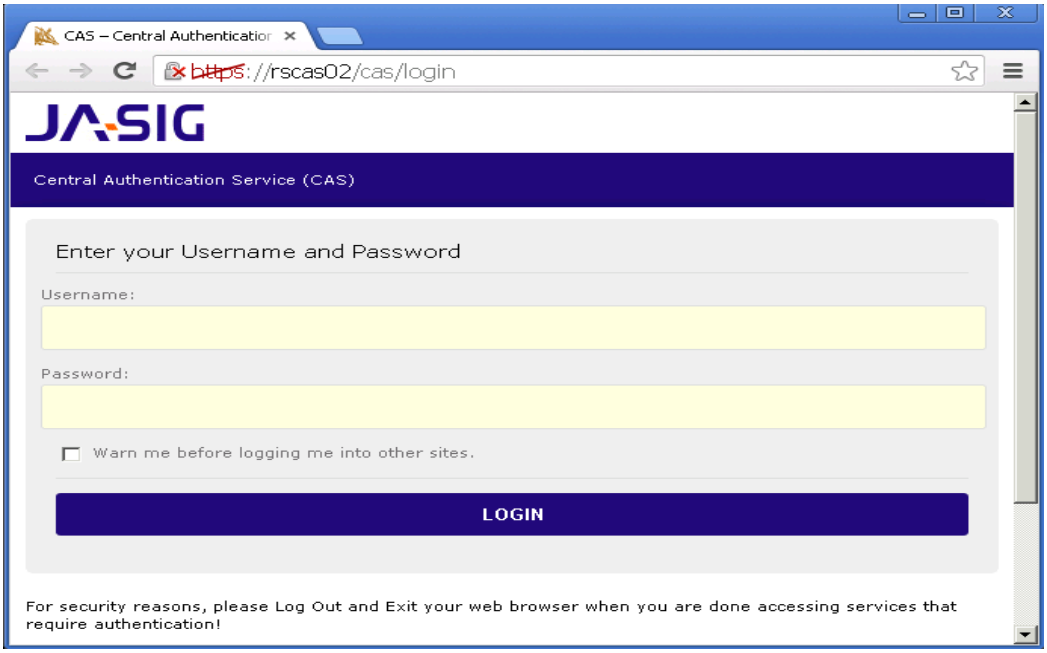
## Customizing CAS Proxy Response

The CAS proxy response file is at: *cas-server-webapp-3.5.2.war/WEB-INF/view/jsp/protocol/2.0/CasServiceValidationSuccess.jsp*. Add following element to the *‹cas: authenticationSuccess›* between *‹cas:user›* and *‹c:if test="${not empty pgtIou}"›*.

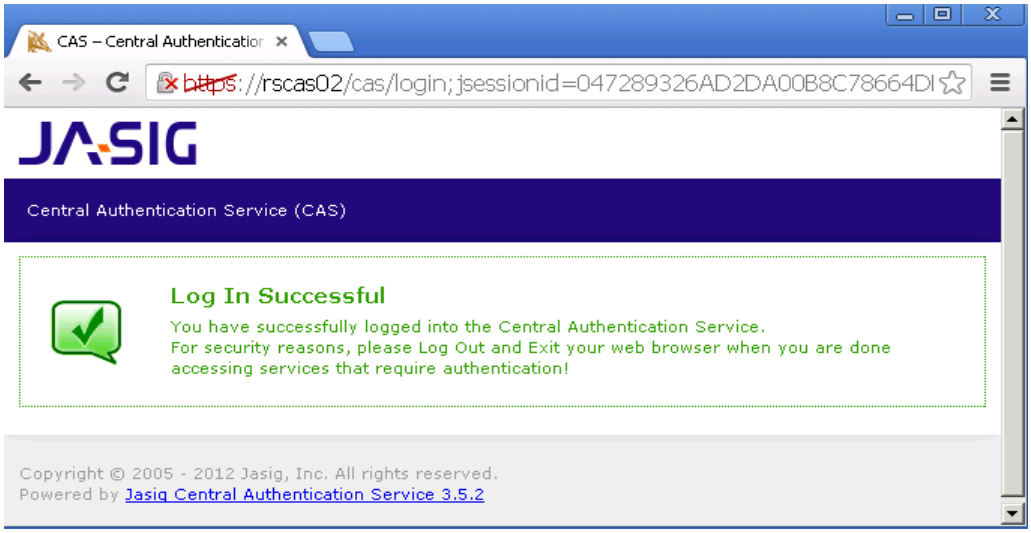
```
<cas:authenticationSuccess>
  <!-- other beans here -->
  </cas:user>
  <!-- Documentum customization begin -->
  <c:forEach var="auth" items="${assertion.chainedAuthentications}">
    <c:forEach var="attr" items="${auth.principal.attributes}" >
    <cas:attribute name="${fn:escapeXml(attr.key)}"   value="${fn:escapeXml(attr.value)}"/>
   </c:forEach>
 </c:forEach>
 <!-- Documentum customization end -->
 <c:if test="${not empty pgtIou}">
</cas:authenticationSuccess>
```

## Validating CAS Deployment

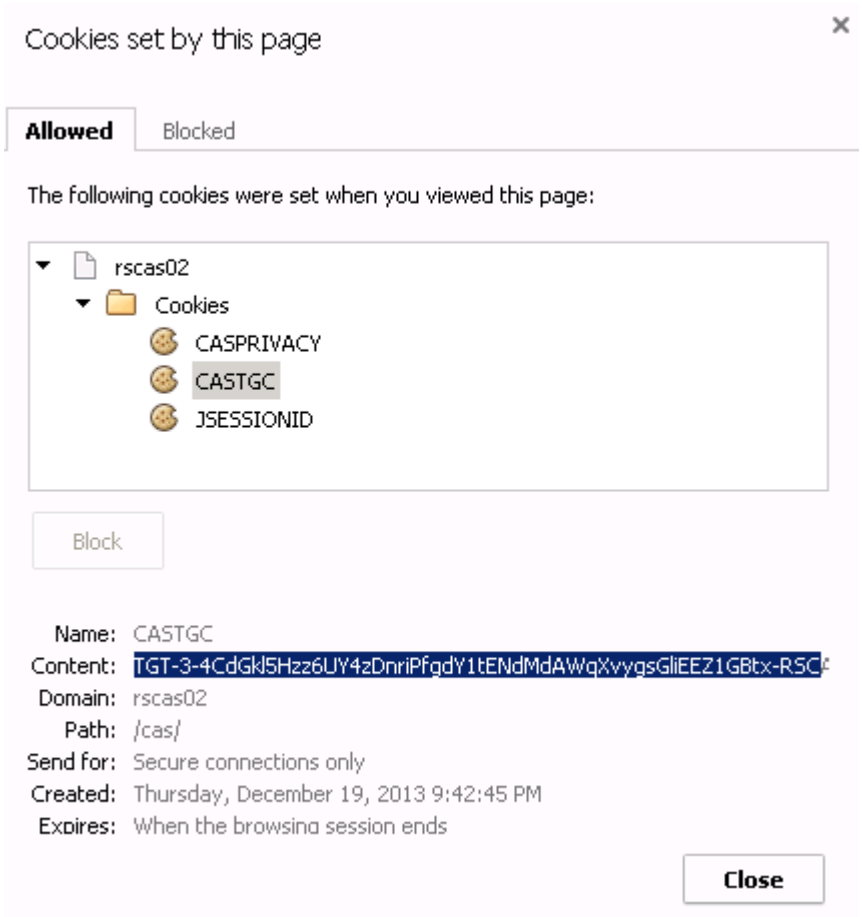
After you complete these the configurations above, deploy the CAS WAR file (renamed to *cas.war*) to the web container and start the web server. The CAS login can be tested by accessing the page: *https//cas-server/cas/login*. A CAS login page should be returned.


CAS – Central Authentication
https://rscas02/cas/login
JA-SIG
Central Authentication Service (CAS)
Enter your Username and Password
Username:
Password:
Warn me before logging me into other sites.
LOGIN
For security reasons, please Log Out and Exit your web browser when you are done accessing services that require authentication!

Enter the user id and password. The login successful page appears.


CAS – Central Authentication
https://rscas02/cas/login;jsessionid=047289326AD2DA00B8C78664DI
JA-SIG
Central Authentication Service (CAS)
Log In Successful
You have successfully logged into the Central Authentication Service.
For security reasons, please Log Out and Exit your web browser when you are done accessing services that require authentication!
Copyright © 2005 - 2012 Jasig, Inc. All rights reserved.
Powered by Jasig Central Authentication Service 3.5.2

Please check the page info. You will find a CASTGC cookie set to the browser client which carries a TGT token. The client reuses this cookie to achieve SSO.



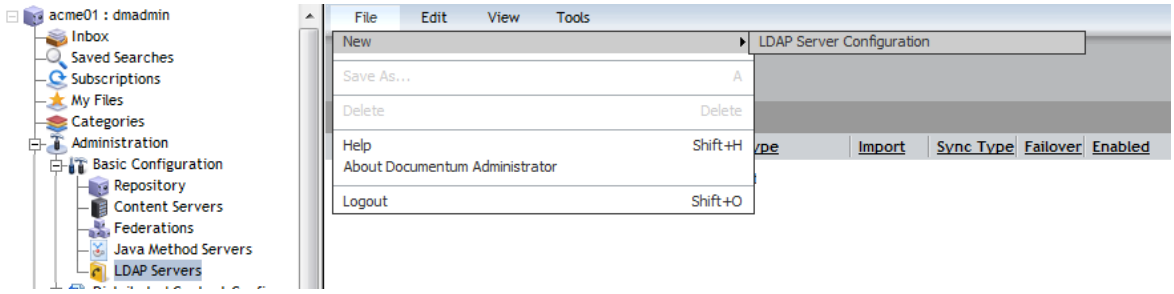
## Content Server Configuration

### Preliminaries

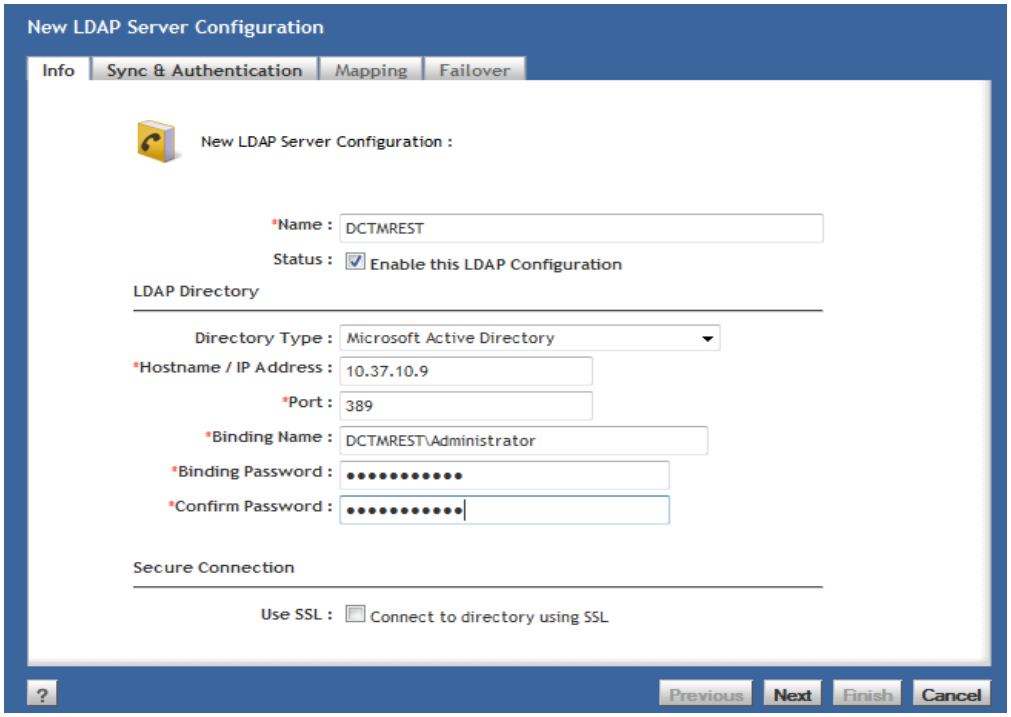
CAS SSO for Content Server is available since Documentum 7.1. Please refer to Content Server Administration Guide for the detailed configuration. In this chapter, we just reference the key steps for Content Server CAS setup.

Prior to the CAS plugin configuration, create an LDAP configuration to synchronize the LDAP users to the Content Server repository (if not yet). Content Server Administration Guide has details on the setup of LDAP configuration. Just for your quick reference, we take the snapshots for the setup.

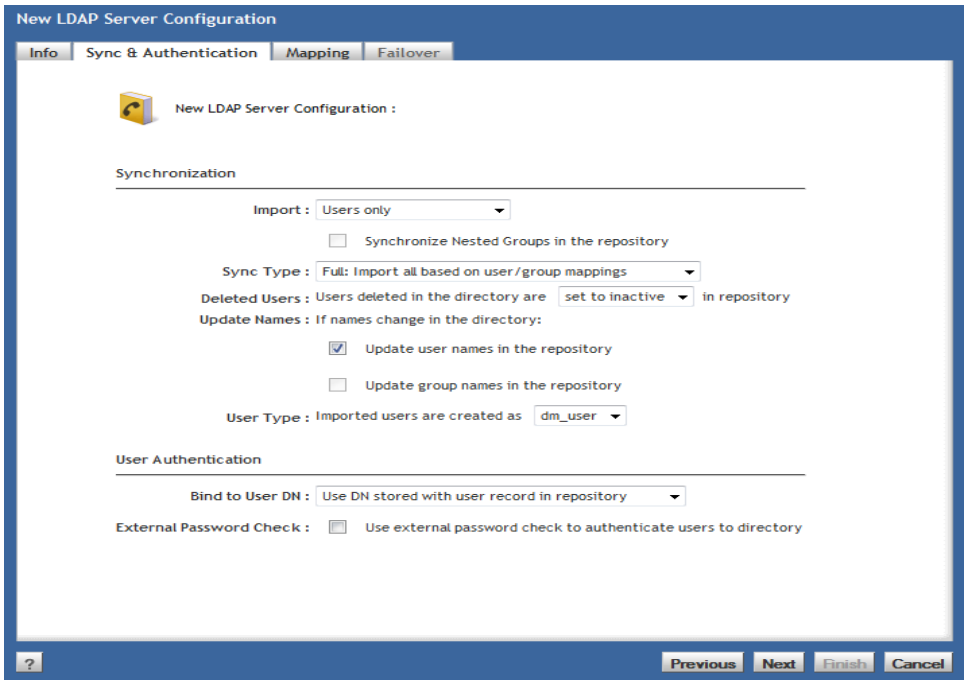
**Step 1):** Login to **Documentum Administrator** and create an **LDAP Server Configuration**.



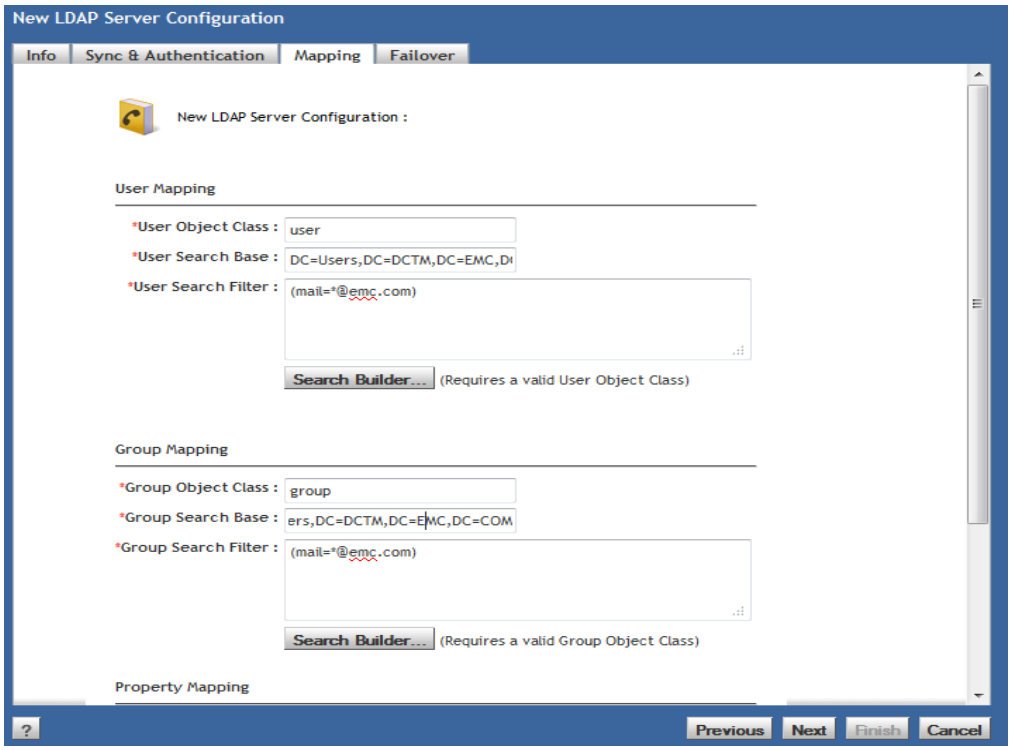

**Step2):** Enter the LDAP name, directory type, host, port and admin credentials. The LDAP configuration name should be the domain name.



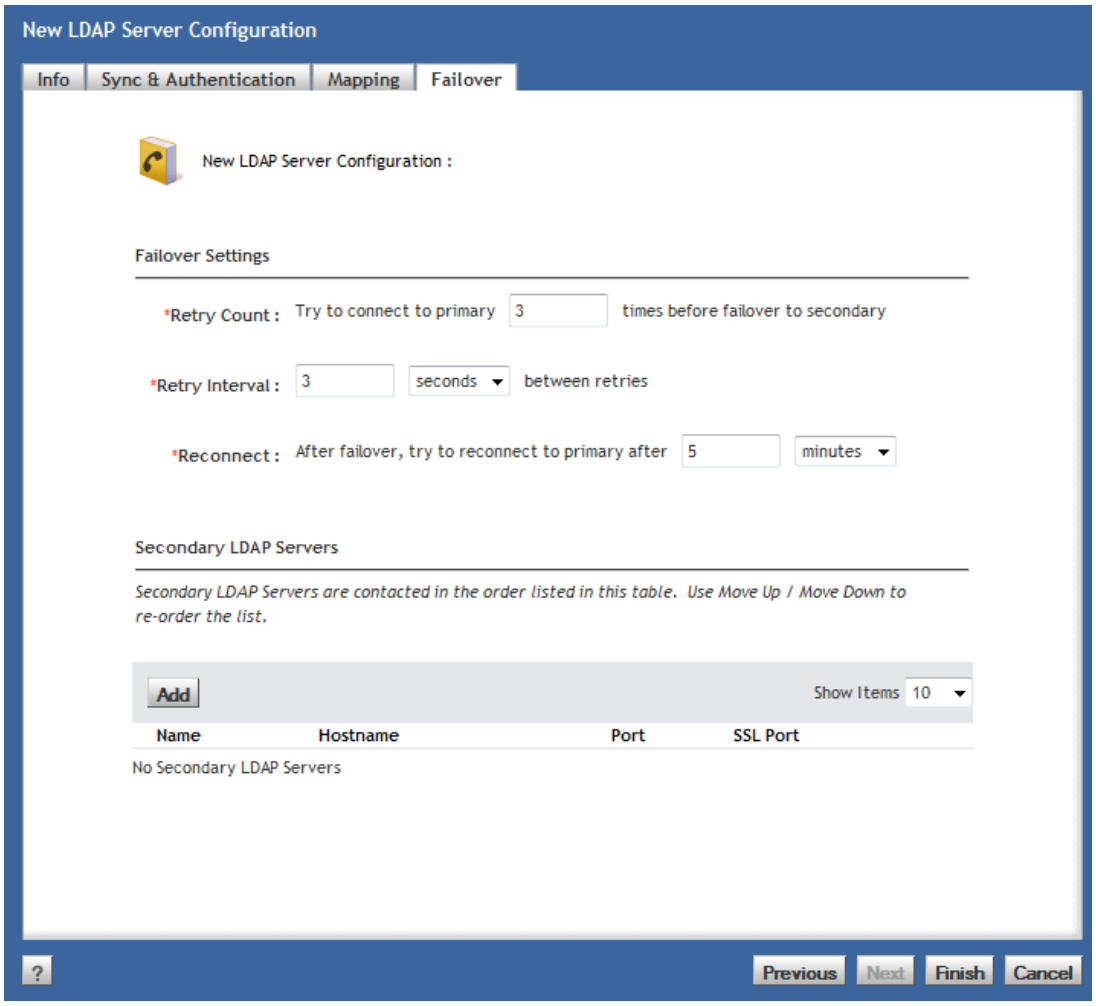
**Step 3):** Select users and/or groups to synchronize.


New LDAP Server Configuration
Info
Sync & Authentication
Mapping
Failover
New LDAP Server Configuration :
Synchronization
Import : Users only
Synchronize Nested Groups in the repository
Sync Type : Full: Import all based on user/group mappings
Deleted Users : Users deleted in the directory are set to inactive in repository
Update Names : If names change in the directory:
Update user names in the repository
Update group names in the repository
User Type : Imported users are created as dm_user
User Authentication
Bind to User DN : Use DN stored with user record in repository
External Password Check : Use external password check to authenticate users to directory
?
Previous
Next
Finish
Cancel

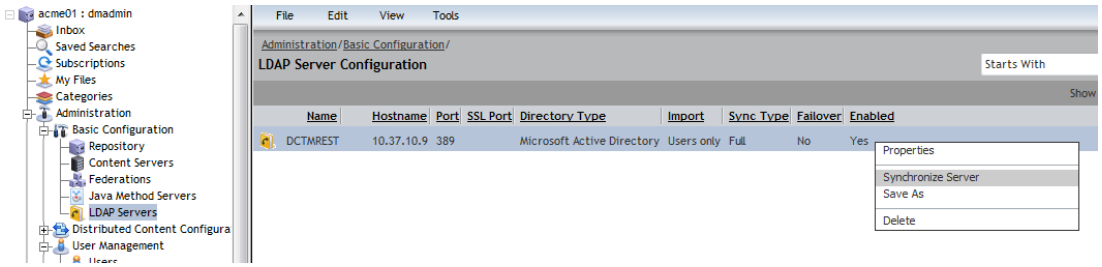
**Step 4):** Map the user class and search base. If necessary, set filters to synchronize a subset of LDAP users to the repository.



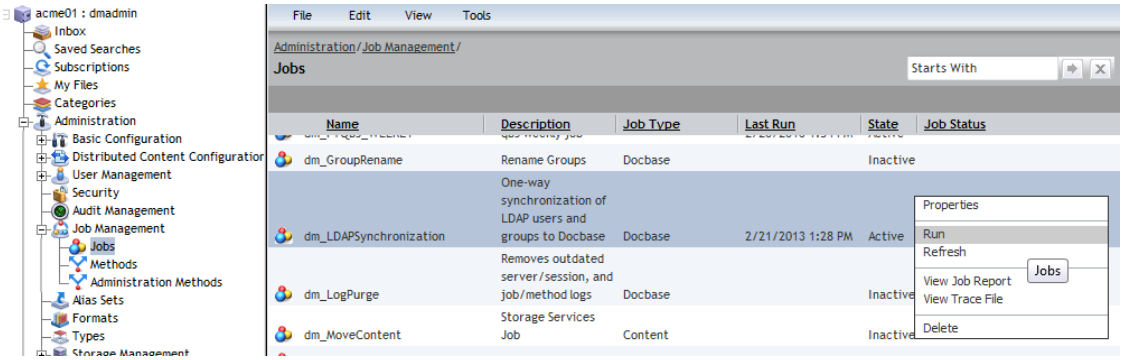
**Step 5):** Configure Failover settings and complete the configuration.




**Step 6):** Synchronize the LDAP server immediately.




**Step 7):** It's better to start a job to synchronize the LDAP periodically.




## Enabling CAS SSO Plugin

### Step 1):

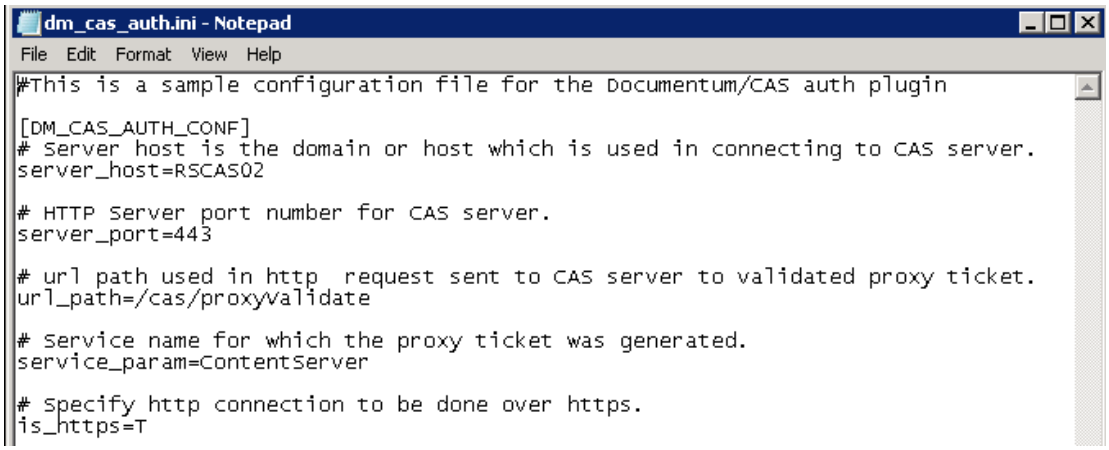
Navigate to the Content Server installation directory and find the CAS plugin DLL file (*dm_cas_auth.dll*). Typically it's located at: *\Documentum\product\7.1\install\external_apps\authplugins\CentralAuthentication Service*

### Step 2):

Copy it to the target directory: *\Documentum\dba\auth*

### Step 3):

Create an empty *dm_cas_auth.ini* file under the same directory, and fill in the following entries:


dm_cas_auth.ini - Notepad
File Edit Format View Help
#This is a sample configuration file for the Documentum/CAS auth plugin
[DM_CAS_AUTH_CONF]
# Server host is the domain or host which is used in connecting to CAS server.
server_host=RSCAS02
# HTTP Server port number for CAS server.
server_port=443
# url path used in http request sent to CAS server to validated proxy ticket.
url_path=/cas/proxyValidate
# Service name for which the proxy ticket was generated.
service_param=ContentServer
# Specify http connection to be done over https.
is_https=T

The *server_host* and *server_port* are for the CAS server deployment.

The *url_path* is defined by CAS servlet mapping. Typically, It's not required to change the path..

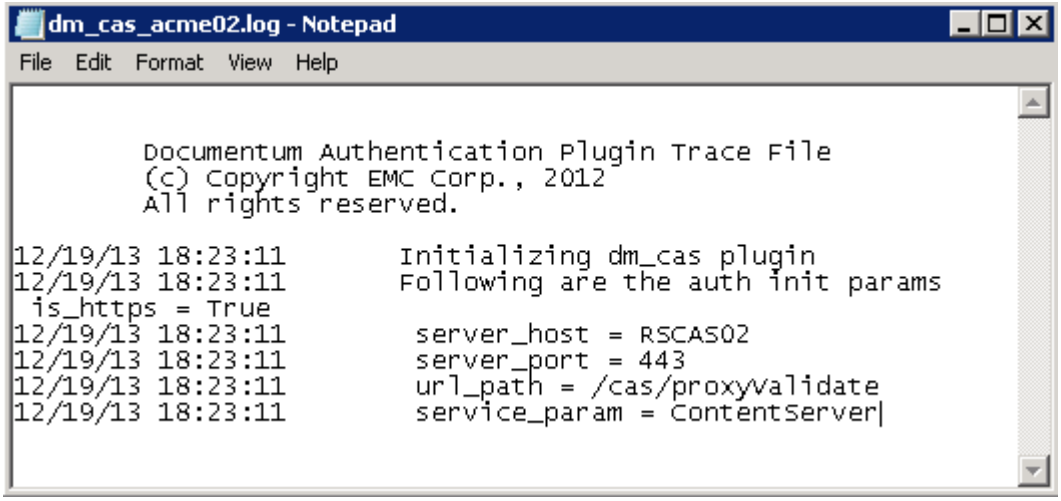
The *service_param* specifies the proxy service name for Content Server. It has to be registered in CAS service registry. Use this value consistently across the CAS server, REST server, and Content Server. The value is not necessary to be changed unless separate proxies have been configured for multiple Content Server instances.

**Step 4):**

Restart the repository.

### Validating Content Server Configuration

Check the repository log under: *Documentum**dba**logs*. The following log information will be desplayed:


dm_cas_acme02.log - Notepad
File Edit Format View Help
Documentum Authentication Plugin Trace File
(c) Copyright EMC Corp., 2012
All rights reserved.
12/19/13 18:23:11 Initializing dm_cas plugin
12/19/13 18:23:11 Following are the auth init params
is_https = True
12/19/13 18:23:11 server_host = RSCAS02
12/19/13 18:23:11 server_port = 443
12/19/13 18:23:11 url_path = /cas/proxyValidate
12/19/13 18:23:11 service_param = ContentServer

For more information about Content Server configuration, please refer to the whitepaper “Documentum Content Server Central Authentication Service (CAS) SSO”.

## REST Server Configuration

### Preliminaries

CAS SSO is available for Documentum REST Services since version 7.1.

To make CAS SSO work for the REST server, the web container which hosts the REST services must enable SSL. For formal product implementation, you need to register the SSL certificate to an authority. For quick startup, create a certificate for your web container with any of the following tools. For instance, Java keytool (http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html), Openssl (http://www.openssl.org/), or any other certificate tool (like Portecle, http://portecle.sourceforge.net/). After the certificate generation, import it to the JRE keystore of the REST server with following command.

```
keytool -import -keystore <path-of-the-jre-cacert> -storepass "<password>" -alias "<rest-alias>" -file
<path-of-the-rest-certificate-file>
```

In addition to that, this certificate must be trusted by the CAS server for ticket validation (mandatory in HTTPS), so **go to CAS server** and import it to the CAS server's JRE trust store.

```
keytool -import -keystore <path-of-the-jre-cacert> -storepass "<password>" -alias "<rest-alias>" -file
<path-of-the-rest-certificate-file>
```

### Enabling CAS Authentication Scheme

CAS is not the default authentication scheme for Documentum REST Services. To enable CAS, open *dctm-rest.war* and navigate to */WEB-INF/Classes/rest-api-runtime.properties*. Open the file, edit it and repackage the war.

```
rest.security.auth.mode=ct-cas
rest.security.cas.server.url=https://cas-server/cas
rest.security.cas.server.login.url=https://cas-server/cas/login
rest.security.cas.server.logout.url=https://cas-server/cas/logout
rest.security.cas.server.tickets.url=https://cas-server/cas/v1/tickets
rest.security.cas.proxy.service=ContentServer
rest.security.server.url=rest-server:8443
rest.security.cas.callback.service.url=https://rest-server:8443/dctm-rest/
rest.security.auth.cas.client.pgt.storage=inmemory
```

#### rest.security.auth.mode

The authentication mode should be "*ct-cas*" which stands for CAS SSO. With this mode, the basic authentication for inline users is disabled. There is another option that enables both HTTP Basic and CAS modes, which will be introduced in section *Multiple Authentication Schemes*.

The CAS server related URLs point to the host of the CAS server. These include

- *rest.security.cas.server.url*
- *rest.security.cas.server.login.url*
- *rest.security.cas.server.logout.url*
- *rest.security.cas.server.tickets.url*

**rest.security.cas.proxy.service**

The proxy service is '*ContentServer*', which has been registered in CAS service registry. It can be changed but we do not recommend doing that unless separate proxy services need to be configured for multiple Content Server instances.

**rest.security.server.url**

The server URL is the REST server's host and port mapping, which is used by the CAS client agent to construct the redirecting-back resource URLs.

**rest.security.cas.callback.service.url**

The CAS callback service URL points to the REST server's CAS proxy callback service. Documentum REST Services has default implementation of the callback service, so it's pointed to the REST server's root context URL. If the customers are familiar with CAS Java client and Spring MVC, they can build their custom CAS callback service and PGTIOU PGT mapping storage.

**rest.security.auth.cas.client.pg.storage**

The CAS PGT storage supports two modes, '*inmemory*' and '*ehcache*'. In-memory is the default mode and is used for single REST server deployment. If customers deploy a cluster environment for REST services, the mode needs to be changed to Ehcache. Section "REST Server Clustering for CAS" will discuss more details about it.

### Validating REST Deployment

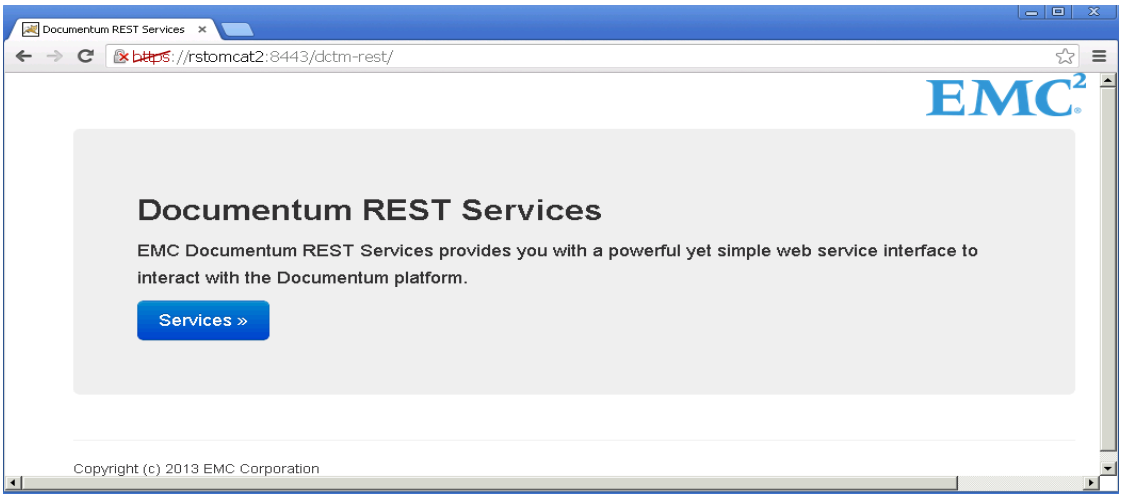
After the WAR file has been configured and deployed, restart the web container. By default (if the logging level is INFO), the following entry is visible in the startup logs.

```
Authentication mode is set to ct-cas for the Documentum Core REST Services.
```

Then try to access the Documentum REST service via a web browser.

**Step 1): Verify home document resource**

Enter the URL in the address bar or your web browser: *http://rest-server:8443/dctm-rest*


Documentum REST Services
https://rstomcat2:8443/dctm-rest/
EMC²
Documentum REST Services
EMC Documentum REST Services provides you with a powerful yet simple web service interface to interact with the Documentum platform.
Services »
Copyright (c) 2013 EMC Corporation

This step confirms the REST services is running.

### Step 2): Verify Repositories resource

Enter the URL in the address bar or your web browser: *http://rest-server:8443/dctm-rest/repositories*

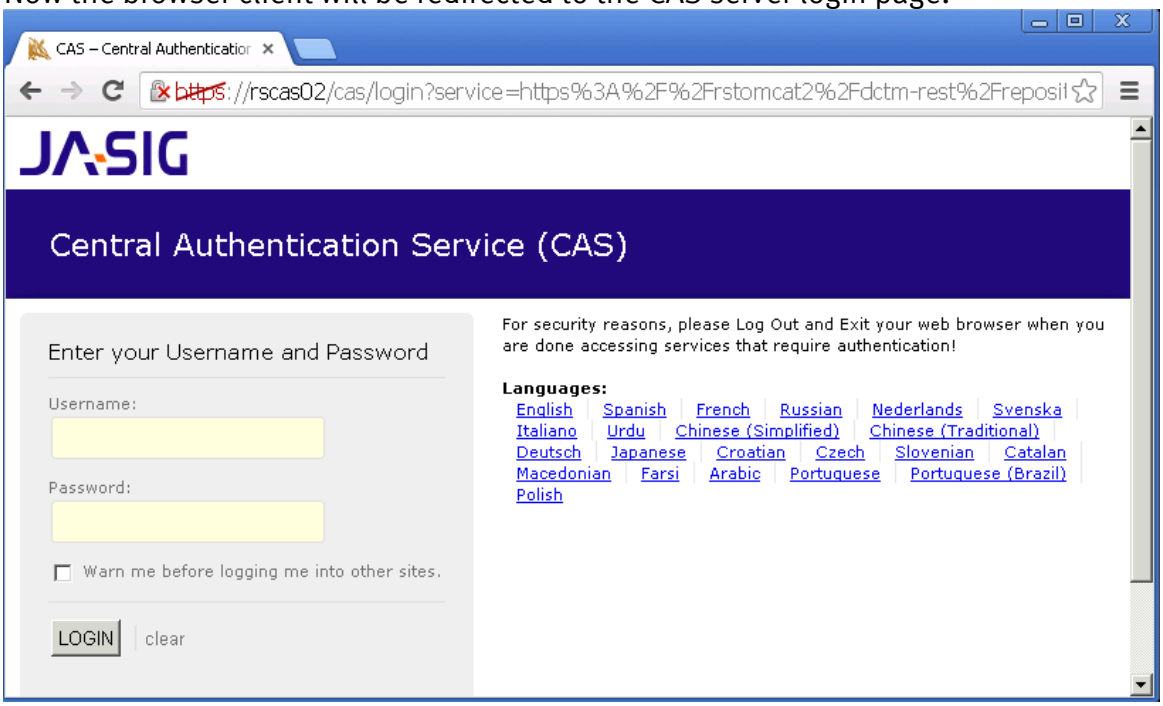
This step confirms the DFC docbroker is working.

### Step 3): Verify Repository resource

The two resources above are for anonymous access, and do not meet any CAS login challenge. Now try with the Repository resource for which the CAS has been enabled.

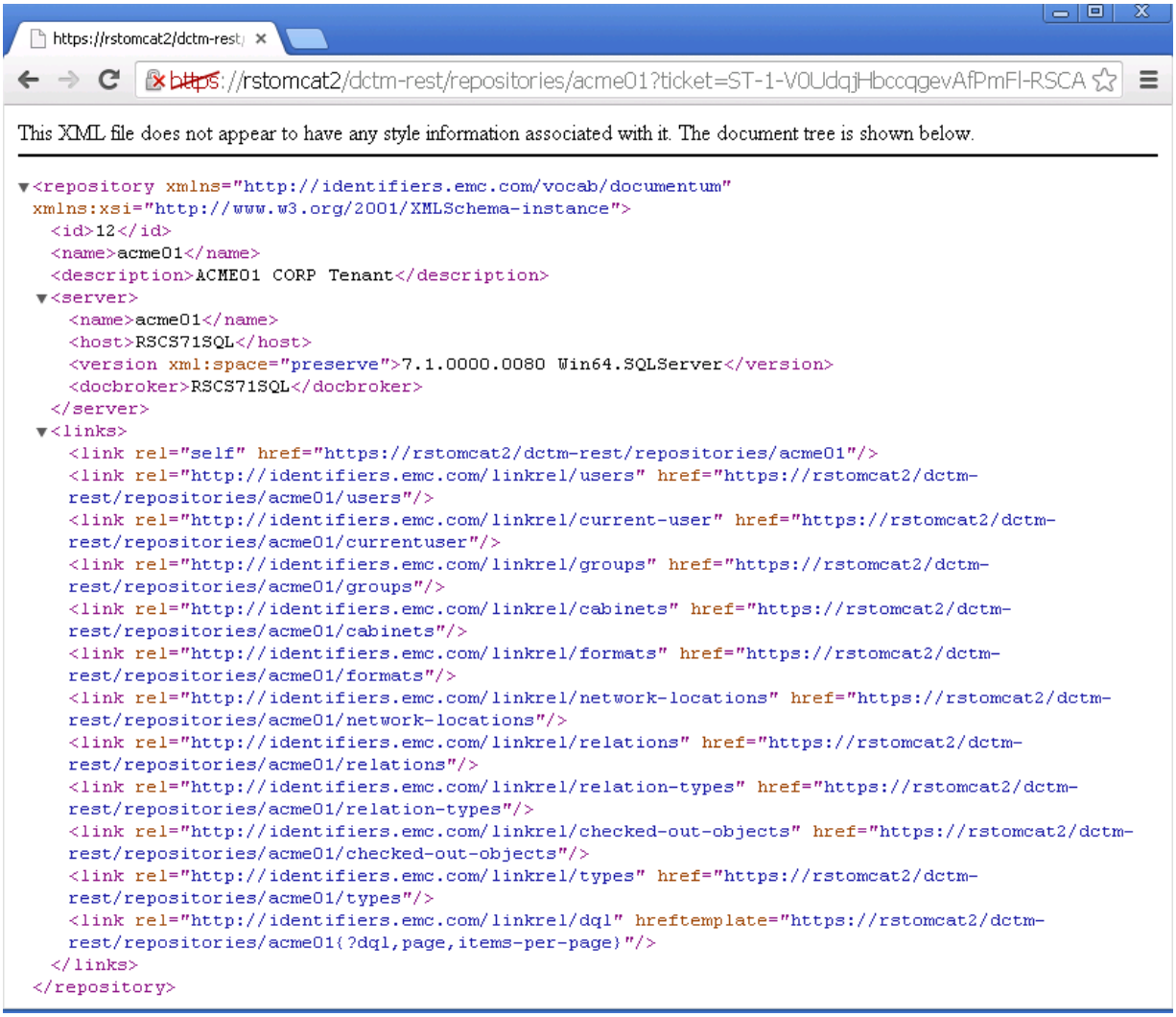
Enter the URL in the address bar of your web browser: *http://rest-server:8443/dctm-rest/repositories/acme01*

Now the browser client will be redirected to the CAS server login page.


CAS – Central Authentication
https://rscas02/cas/login?service=https%3A%2F%2Frstomcat2%2Fdctm-rest%2Freposit
JA-SIG
Central Authentication Service (CAS)
Enter your Username and Password
Username:
Password:
Warn me before logging me into other sites.
LOGIN
clear
For security reasons, please Log Out and Exit your web browser when you are done accessing services that require authentication!
Languages:
English | Spanish | French | Russian | Nederlands | Svenska | Italiano | Urdu | Chinese (Simplified) | Chinese (Traditional) | Deutsch | Japanese | Croatian | Czech | Slovenian | Catalan | Macedonian | Farsi | Arabic | Portuguese | Portuguese (Brazil) | Polish

Please note the redirected URL that the original Repository resource URL is appended as a 'service' parameter to the CAS login URL.

Now please enter the user name and password. The browser will be redirected back to the original Repository resource after a successful CAS login.




Please note the redirected URL that a ST is appended to it in parameter 'ticket' which was sent by CAS. Please note that an ST, which is sent from the CAS server, is appended to the redirected URL in the ticket parameter.

With these steps, the CAS SSO for Documentum REST Services has been working!

## Advanced CAS Integration Options

This chapter introduces some advanced deployment options for the Documentum REST Services with CAS SSO.

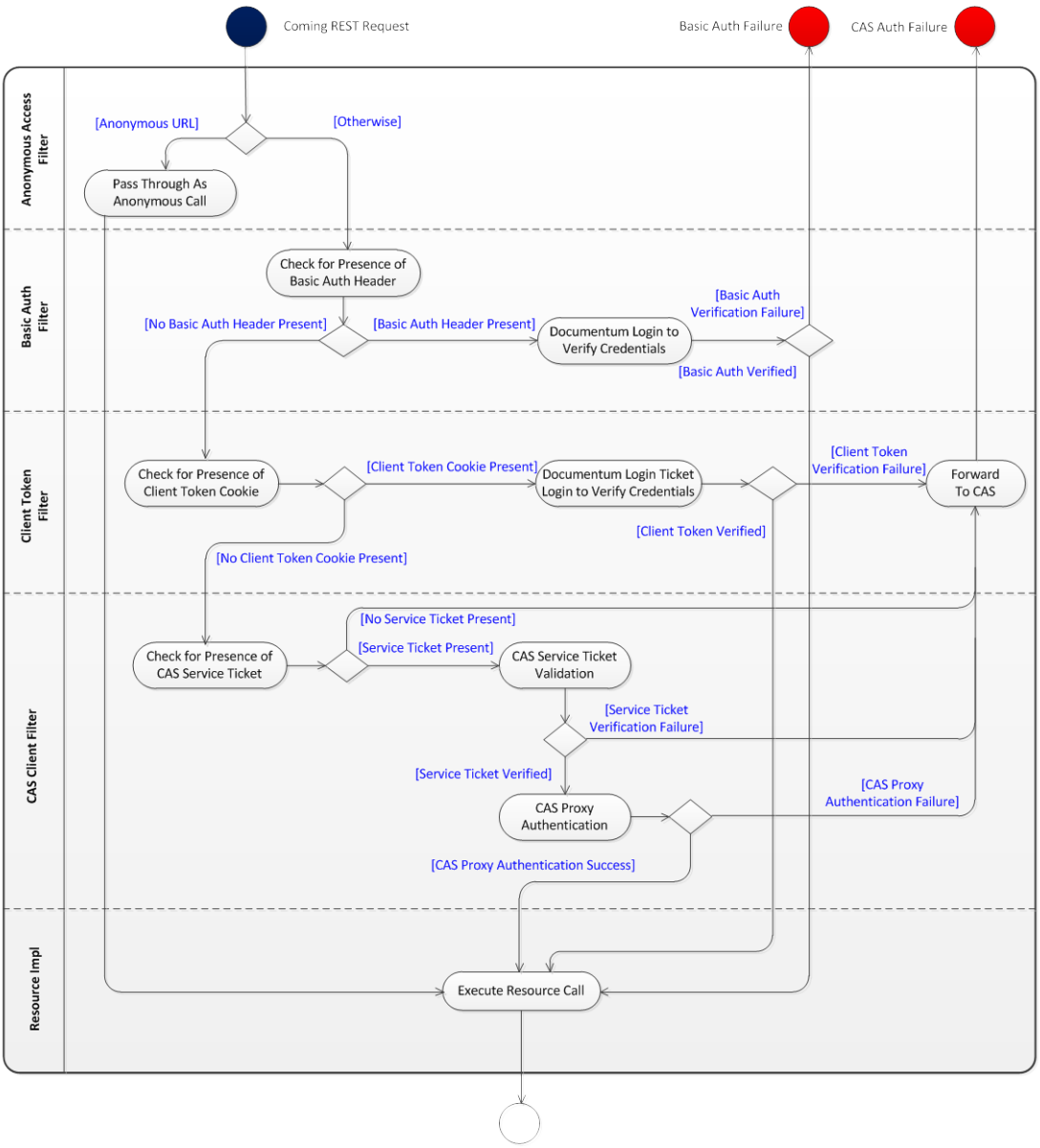
### Multiple Authentication Schemes

Documentum REST Services allows for multiple authentication schemes in your production environments. Various combinations of authentication schemes are configured by modifying *rest.security.auth.mode* in the *rest-api-runtime.properties* file. The following CAS–related combinations of authentication schemes are supported:

- HTTP Basic and CAS with client tokens (*rest.security.auth.mode=basic-ct-cas*)
- CAS with client tokens (*rest.security.auth.mode=ct-cas*)

The following diagram illustrates the workflow of authentication when a request comes to access a resource where both HTTP Basic and CAS are working.

Figure 7 Basic and CAS Authentication Work Together




Here we show samples of REST messages for multi-authentication schemes.

## Example 1 No Authorization header

```
// request
GET /${resource-url} HTTP/1.1
// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
WWW-Authenticate: CAS <MY_REALM>
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"Full authentication is required to access this resource"
}
```

### Example 2 Mismatching Authorization header

```
// request
GET /${resource-url} HTTP/1.1
Authorization: CAS <MY_REALM>
// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
WWW-Authenticate: Negotiate
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"Full authentication is required to access this resource"
}
```

### Example 3 Matching basic Authorization header

```
// request
GET /${resource-url} HTTP/1.1
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ
// response
HTTP/1.1 200 OK
// resource body
Example 4-13. Matching CAS ticket
// request
GET /${resource-url}?ticket=ST-29-HeJcl956dMmTttZhLBPZ-CAS.EMC.COM HTTP/1.1
// response
HTTP/1.1 200 OK
// resource body
```

### Example 4 Matching basic Authorization header with bad credential

```
// request
GET /${resource-url} HTTP/1.1
Authorization: Basic ZG1hZG1pbjpwYXNzd29yZ++bad++credential
// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic <MY_REALM>
{
"status":401,
"code":"E_BAD_CREDENTIALS_ERROR",
"message":"Authentication failed because an invalid credential is provided.",
"details":"(DM_SESSION_E_AUTH_FAIL) error: \"Authentication failed for user
badboy with docbase space01.\""
}
```

### Example 5 Matching CAS ticket with bad credential (no redirect)

```
// request
GET /${resource-url}?ticket=ST-29-HeJcl956dMmTttZhLBPZ++bad++credential HTTP/1.1
DOCUMENUM-NO-CAS-REDIRECT: true
// response
HTTP/1.1 401 Unauthorized
WWW-Authenticate: CAS <MY_REALM>
{
"status":401,
"code":"E_GENERAL_AUTHENTICATION_ERROR",
"message":"Authentication failed.",
"details":"ticket 'ST-29-HeJcl956dMmTttZhLBPZ++bad++credential' not recognized."
}
```

### Example 6 Matching CAS ticket with bad credential (redirect)

```
// request
GET /${resource-url}?ticket=ST-29-HeJcl956dMmTttZhLBPZ++bad++credential HTTP/1.1
DOCUMENUM-NO-CAS-REDIRECT: false
// response
HTTP/1.1 302 Moved
Location: https://cas-server/cas/login
```

## CAS SSO across Content Server Repositories

The Content Server CAS plugin is enabled per repository. To enable CAS SSO for multiple repositories, perform following actions:

- Configure the LDAP server for each repository
- Enable the CAS SSO plugin for each repository
- Configure repository trust

The trust relationship setup for Content Server repositories can be found in Content Server Administration Guide. The reason to require trust across repositories is because the CT cookie is depending on the Documentum Login Ticket (LT) and the LT is by default private to one repository. To make the CT cookie usable for all repositories, please setup trust across these repositories. The trust across multiple repositories can be setup using DFC API. Here we show samples about how to import the LT keys between two repositories.

### Example 7 DFC Samples to Setup Trusts Between Repositories

```
String exportedLTK = session1.exportTicketKey("password");
System.out.println("Exported LTK from repository acme01:" + exportedLTK);
boolean imported = session2.importTicketKey(exportedLTK, "password");
System.out.println("Imported LTK to repository acme02: " + imported);
```

```
IDfTypedObject docbaseConfig1 = session1.getDocbaseConfig();
docbaseConfig1.setRepeatingString("trusted_docbases", 0, "acme02");
((IDfSysObject) docbaseConfig1).save();
for(int k=0; k<docbaseConfig1.getAttrCount(); k++) {
    System.out.println(docbaseConfig1.getAttr(k).getName() + "\t=" + docbaseConfig1.getValueAt(k));
}
IDfTypedObject docbaseConfig2 = session2.getDocbaseConfig();
docbaseConfig2.setRepeatingString("trusted_docbases", 0, "acme01");
((IDfSysObject) docbaseConfig2).save();
for(int k=0; k<docbaseConfig2.getAttrCount(); k++) {
    System.out.println(docbaseConfig2.getAttr(k).getName() + "\t=" + docbaseConfig2.getValueAt(k));
}
```

## Security Configuration for Client Token

By default, client tokens are encrypted by the RSA JSafeJCE provider. To support flexible security options, Documentum REST Services allows the use of different cryptography algorithms to encrypt and decrypt CTs. All options are available in the *rest-api-runtime.properties* file.

```
rest.security.crypto.algorithm=
rest.security.crypto.algorithm.parameters.class=
rest.security.key.algorithm=
rest.security.random.algorithm=
rest.security.crypto.provider=
rest.security.crypto.provider.class=
rest.security.crypto.key.size=
rest.security.crypto.block.size=
rest.security.crypto.key.salt=
rest.security.crypto.provider.jsafejce.mode=
```

Currently, we support two crypto providers:

- JsafeJCE (RSA provider, the default provider)
- BC (Bouncy Castle provider, the alternate provider)

The default crypto algorithm is AES128. It could be changed to other algorithms like DESede, RC5, etc.

***Note:*** *The default cryptography algorithms used for the client token encryption and decryption are strong enough in most cases. Therefore, we recommend keeping the original settings unless special security requirements are required for your*

*organization. It's strongly recommended to consult your security department and EMC support to make crypto changes for the CT token.*

The crypto key size and block size can be changed, too, but please make sure it meets the requirement of specific crypto algorithm. Also, please note that the default version of Java Cryptography Extension (JCE) policy files bundled in the JDK(TM) environment limits the key size of cryptography algorithms to 128 bits. To remove this restriction, download Unlimited Strength Jurisdiction Policy Files from the Oracle web site.

The default security provider works in FIPS140_MODE (http://csrc.nist.gov/groups/STM/cmvp/standards.html). To support security compatibility for specific environment, the provider may have to be run in NON_FIPS140_MODDE.

***Note:*** *One known issue about FIPS140 is for IBM WebSphere Server that it can be run only in NON_FIPS140_MODE. So make sure to change the mode to NON_FIPS140_MDE for the REST server deployment in WebSphere to support CAS SSO.*

### REST Server Clustering for CAS

REST servers can be deployed in a clustered environment where there usually has a reverse proxy server deployed in front. To deploy REST server clustering, perform the following steps:

**Step 1):**

Update the key salt to a non-empty value in all *rest-api-runtime.properties* files:

```
# Crypto salt for client token encryption and decryption
# For a multi-node deployment of REST servers, this property MUST be consistently set across all REST servers.
# For a single-node deployment of REST servers, this property is optional.
# The value CAN be any ascII characters. We recommend that you specify a text no less than 8 characters.
rest.security.crypto.key.salt=
```

Step 2):

Update the PGT storage type to *ehcache* in all *rest-api-runtime.properties*:

```
# Specifies the place where CAS clients store and retrieve PGT by mapping them to a specific
ProxyGrantingTicketIou. This property MUST be specified with a non-empty value.

# Valid values:
#    1) inmemory
#    2) ehcache
```

During the PT negotiation between the REST server and CAS server, the CAS server has to make a callback to the REST server requesting the PGT. If REST servers are deployed in a cluster, the callback may not find the REST server requesting the PGT. Therefore, all REST servers must maintain the same PGT IOU/PGT mappings.

To do this, REST servers utilizes Ehcache to perform the replication of PGT IOU/PGT mappings across the cluster.

When using ehcache storage, configure the *dctm-rest.war\WEB-INF\classes\ehcache-cas.xml* file:

```
<ehcache xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="false">
  <defaultCache timeToIdleSeconds="100" timeToLiveSeconds="100" />
  <!-- Choose either multicast or manual for peer discovery -->
  <cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
properties="peerDiscovery=manual,rmiUrls=//${another_peer_ip_address}:40001/com.emc.document
um.rest.security.cache.EhcacheBackedOneTimeProxyGrantingTicketStorageImpl.cache|//${yet_anothe
r_peer_ip_address}:40001/com.emc.documentum.rest.security.cache.EhcacheBackedOneTimeProxyGr
antingTicketStorageImpl.cache" />
  <!-- Change the host name to your host name or IP address, this is important in particular for a
machine in a multihomed environment -->
  <cacheManagerPeerListenerFactory
          class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
          properties="hostName=${change_with_your_host_or_ip},port=40001" />
  <cache
name="com.emc.documentum.rest.security.cache.EhcacheBackedOneTimeProxyGrantingTicketStorag
eImpl.cache"
    maxElementsInMemory="100"
    eternal="false"
    timeToIdleSeconds="100"
    timeToLiveSeconds="100"
    overflowToDisk="false">
  <cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
properties="replicateAsynchronously=false,port=40001"/>
  </cache>
</ehcache>
```

The host names and ports for this REST server and other REST servers in this XML file must be updated.

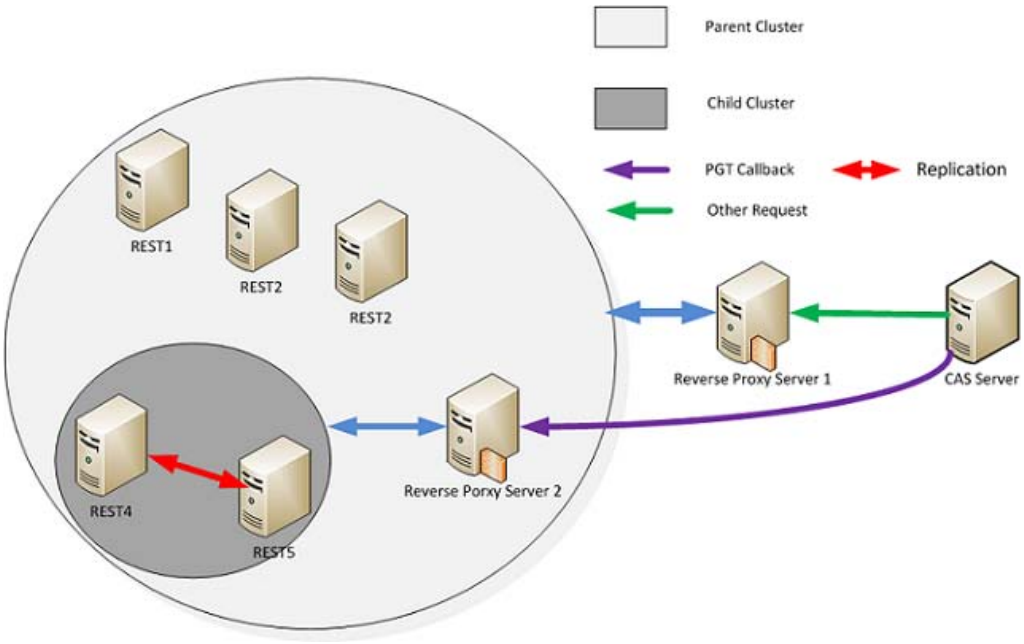
Please note that this sample gives an example of RMI manual peer discovery. It can be changed to other replication methods like JMS or JGroup. To get more options, please refer to: http://ehcache.org/documentation/replication

### Performance Consideration

The replication of PGT IOU/PGT mappings across the whole cluster may degrade performance, especially when the number of REST server is large. In this case, create a child cluster behind a reverse proxy server and limit the replication of PGT IOU/PGT mappings within the child cluster. Next, designate the child cluster to handle PGT callbacks. This method reduces the amount of replication and thus improves performance.

The following diagram illustrates the network topology of this method.

Figure 8 Setup Sub Group for PGT Storage




This method requires the following configurations:

- For all REST servers in the parent cluster, the callback URL must be set to the address of the reverse proxy server that is placed in front of the child cluster. (In the diagram, Reverse Proxy Server 2)
- Peer discovery for all REST servers must be configured in the child cluster in which all these REST servers must be set as peers.

## CAS Server Clustering

The CAS authentication can be deployed in a clustered environment to achieve high availability (HA). Please the instructions on Jasig wiki to configure CAS clustering: https://wiki.jasig.org/display/CASUM/Clustering+CAS

If CAS clustering utilizes Ehcache to make all nodes in the cluster recognize and validate each other's tickets, make the following modifications:

In *cas.war\WEB-INF\spring-configuration\ticketRegistry.xml*, set *shared* of the *cacheManager* bean to true.

```
<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
   <!-- Documentum customization start -->
   <property name="configLocation" value="classpath:ehcache-replicated.xml" />
   <property name="shared" value="true" />
   <!-- Documentum customization end -->
   <property name="cacheManagerName" value="ticketRegistryCacheManager" />
 </bean>
```

The default Ehcache configuration recommends setting TGT/PGT replication in *async* mode. However, in RESTful Services, an ST/PT ticket request may happen immediately after the TGT/PGT generation (in milliseconds). Therefore, it is strongly recommended using *sync* mode for both ST and TGT replications.

```
<bean id="ticketGrantingTicketsCache"
class="org.springframework.cache.ehcache.EhCacheFactoryBean"
   parent="abstractTicketCache">
   <property name="cacheName" value="org.jasig.cas.ticket.TicketGrantingTicket" />
   <property name="cacheEventListeners">
   <!-- Documentum customization start -->
    <ref local="ticketRMISynchronousCacheReplicator" />
   <!-- Documentum customization end -->
   </property>
   <!-- The maximum number of seconds an element can exist in the cache without being accessed. The
element expires at this limit and will no longer be returned from the cache. The default value is 0,
which means no TTI eviction takes place (infinite lifetime). -->
   <property name="timeToIdle" value="0" />
   <!-- The maximum number of seconds an element can exist in the cache regardless of use. The
element  expires at this limit and will no longer be returned from the cache. The default value is 0,
which means  no TTL eviction takes place (infinite lifetime). -->
   <property name="timeToLive" value="0" />
 </bean>
```

To make the CAS ticket replication work, enable the cache replication engine from *cas.war\WEB-INF\classes\ehcache-replicated.xml*. Here is the example of RMI replicator.

### RMI Manual Peer Discovery

```
<ehcache name="ehCacheTicketRegistryCache"
  updateCheck="false"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">
  <diskStore path="java.io.tmpdir/cas"/>
   <!-- Documentum customization start -->
  <!-- manual RMI peer discovery -->
  <cacheManagerPeerProviderFactory
    class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
properties="peerDiscovery=manual,rmiUrls=//${the_other_cas_server}:${the_other_cas_server_cache
_port}/org.jasig.cas.ticket.ServiceTicket|//${the_other_cas_server}:${the_other_cas_server_cache_por
t} /org.jasig.cas.ticket.TicketGrantingTicket" />
  <cacheManagerPeerListenerFactory
       class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
properties="hostName=${this_cas_server},port=${this_cas_server_cache_port},remoteObjectPort=${t
he_other_cas_server_cache_port} " />
    <!-- Documentum customization start -->
</ehcache>
```

The Ehcache site provides more options for cache replication, http://ehcache.org/documentation/replication and https://wiki.jasig.org/display/CASUM/EhcacheTicketRegistry

### RMI Multicast Peer Discovery

Here is a sample configuration in the *ehcache-replicated.xml* file to set automatic peer discovery using the same RMI protocol.

```
<ehcache name="ehCacheTicketRegistryCache"
  updateCheck="false"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">
  <diskStore path="java.io.tmpdir/cas"/>
  <!-- Documentum customization start -->
  <!-- automatic RMI peer discovery -->
  <cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
properties="peerDiscovery=automatic, multicastGroupAddress=230.0.0.1, multicastGroupPort=4446,
timeToLive=32" propertySeparator="," />
    <cacheManagerPeerListenerFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
properties="hostName=${the_other_cas_server}, port=40001, socketTimeoutMillis=2000"/>
    <!-- Documentum customization start -->
</ehcache>
```

### JGroup Peer Discovery

Here is a sample of JGroup cache replication. To use JGroup, please update configurations in both *ticketRegistry.xml* and *ehcache-replicated.xml*.

In *ticketRegistry.xml*:

```
<bean id="ticketGrantingTicketsCache"
class="org.springframework.cache.ehcache.EhCacheFactoryBean"
   parent="abstractTicketCache">
   <property name="cacheName" value="org.jasig.cas.ticket.TicketGrantingTicket" />
   <property name="cacheEventListeners">
   <!-- Documentum customization start -->
    <ref local="ticketjgroupsSynchronousCacheReplicator" />
   <!-- Documentum customization end -->
   </property>
   <!-- The maximum number of seconds an element can exist in the cache without being accessed. The
element expires at this limit and will no longer be returned from the cache. The default value is 0,
which means no TTI eviction takes place (infinite lifetime). -->
   <property name="timeToIdle" value="0" />
   <!-- The maximum number of seconds an element can exist in the cache regardless of use. The
element  expires at this limit and will no longer be returned from the cache. The default value is 0,
which means  no TTL eviction takes place (infinite lifetime). -->
   <property name="timeToLive" value="0" />
</bean>
<!-- Documentum customization start -->
<bean id="ticketjgroupsSynchronousCacheReplicator"
class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicator">
           <constructor-arg name="replicatePuts" value="true"/>
           <constructor-arg name="replicateUpdates" value="true"/>
           <constructor-arg name="replicateUpdatesViaCopy" value="true"/>
           <constructor-arg name="replicateRemovals" value="true"/>
 </bean>
<!-- Documentum customization end -->
```

In *ehcache-replicated.xml*:

```
<ehcache name="ehCacheTicketRegistryCache"
 updateCheck="false"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">
 <diskStore path="java.io.tmpdir/cas"/>
 <!-- Documentum customization start -->
 <cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"
properties="connect=UDP:PING:MERGE2:FD_SOCK:VERIFY_SUSPECT:pbcast.NAKACK:UNICAST:pbcast.
STABLE:FRAG:pbcast.GMS" propertySeparator="::"/>
   <!-- Documentum customization start -->
</ehcache>
```

### Performance Consideration

The CAS clustering and CAS ticket replication is out-of-box functionality which is provided by 3rd party libraries. In REST deployment, customers have the choice to use any CAS ticket replication implementation. However, the replicate method of RMI manual discovery is found not performing well when some REST servers were down. EMC engineering test shows that JGroup and RMI multicast has better performance.

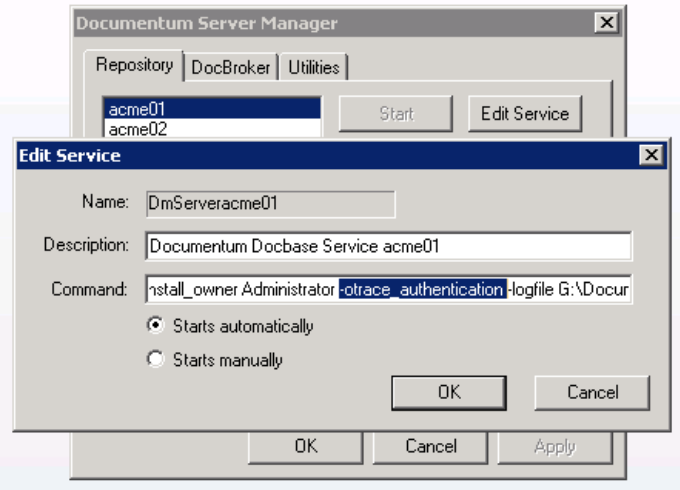
## Logging and Troubleshooting

The CAS SSO for Documentum REST Services involves multiple parties, so it's necessary to collect logs from all parties to get a thorough analysis. Basically, logs need to be collected from these servers:

- Content Server
- REST server
- CAS server
- Cluster (if any)
- LDAP server (optional)

### Content Server Logging

The authentication related logs can be found in the repository log files, typically located in: *\Documentum\dba\log\<repository_name>.log* and *\Documentum\dba\log\dm_cas_<repository_name>.log*. To get the full tracing logs, please enable Content Server tracings by adding "*-otrace_authentication*" in the repository service script.




When the CAS authentication successfully completes, log information that resembles the following will be generated in *dm_cas_<<repository_name>.log* as:

```
12/19/13 18:45:24          Start-Authentication-boblee: userName=boblee, userOsDomain=,
userLdapDn=CN=boblee,CN=Users,DC=ACME,DC=COM
12/19/13 18:45:24          Proxy ticket: ST-2-sAFAvL1WSIZ1BgJgYNFB-RSCAS02.EMC.COM
12/19/13 18:45:24          Sending curl request to following url
https://RSCAS02:443/cas/proxyValidate?service=ContentServer&ticket=ST-2-sAFAvL1WSIZ1BgJgYNFB-
RSCAS02.EMC.COM
12/19/13 18:45:27          Respose code : 200
12/19/13 18:45:27          Respose content :

<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
 <cas:authenticationSuccess>
  <cas:user>boblee</cas:user>
  <cas:attribute name="dmCSLdapUserDN" value="CN=boblee,CN=Users,DC=ACME,DC=COM"/>
  <cas:proxies
   <cas:proxy>https://rstomcat2:8443/dctm-rest/cas/proxy/receptor</cas:proxy>
  </cas:proxies>
 </cas:authenticationSuccess>
</cas:serviceResponse>

12/19/13 18:45:27          Validate Cas user Suceeded
```

And more log information for the successful authentication will be obtained from ‹*repository_name*›*.log* as:

```
2013-12-19T18:45:24.710000      4776[3312]      0100000c8000dd05      [AUTH]  Start-
authenticateByPlugin: Start-authenticateByPlugin: UserLogonName(boblee), PluginId(dm_cas)
2013-12-19T18:45:27.045000      4776[3312]      0100000c8000dd05      [AUTH]  End-
authenticateByPlugin: 1
2013-12-19T18:45:27.045000      4776[3312]      0100000c8000dd05      [AUTH]  End-
AuthenticateByPassword: 1
2013-12-19T18:45:27.051000      4776[3312]      0100000c8000dd05      [AUTH]  Create Session
Log for user : boblee, FILE NAME : 0100000c8000dd05
2013-12-19T18:45:27.052000      4776[3312]      0100000c8000dd05      [AUTH]  Done creating
Session Log File for user : boblee
2013-12-19T18:45:27.177000      4776[3312]      0100000c8000dd05      [AUTH]  Running
dm_checkoutLicense method with arguments: -user_name "boblee" -feature_name "Content_Server"
-feature_version 7.1 -domain ACME
2013-12-19T18:45:27.401000      4776[3312]      0100000c8000dd05      [AUTH]  Final Auth
Result=T, LOGON_NAME=boblee, AUTHENTICATION_LEVEL=8, OS_LOGON_NAME=Administrator,
OS_LOGON_DOMAIN=RSTOMCAT2, CLIENT_HOST_NAME=ausalawrer2m1.corp.emc.com,
CLIENT_HOST_ADDR=192.168.0.5, USER_LOGON_NAME_RESOLVED=1, AUTHENTICATION_ONLY=0,
USER_NAME=boblee, USER_OS_NAME=boblee, USER_LOGIN_NAME=boblee,
USER_LOGIN_DOMAIN=ACME, USER_EXTRA_CREDENTIAL[0]=, USER_EXTRA_CREDENTIAL[1]=,
USER_EXTRA_CREDENTIAL[2]=f0, USER_EXTRA_CREDENTIAL[3]=, USER_EXTRA_CREDENTIAL[4]=,
USER_EXTRA_CREDENTIAL[5]=, SERVER_SESSION_ID=0100000c8000dd05, AUTH_BEGIN_TIME=Thu
Dec 19 18:45:24 2013, AUTH_END_TIME=Thu Dec 19 18:45:27 2013, Total elapsed time=3 seconds
```

### REST Server Logging

The REST server logging is configured in both *dctm-rest.war\WEB-INF\classes\log4j.properties* and *dctm-rest.war\WEB-INF\classes\rest-api-runtime.properties*.

In *log4j.properties* file, please enable tracing level logging for the following Java packages.

```
# Set root logger to INFO and add an appender called A1.
log4j.rootLogger=INFO, A1, R

log4j.logger.com.emc.documentum.rest.log=DEBUG
log4j.logger.com.emc.documentum.rest.dfc=TRACE
log4j.logger.com.emc.documentum.rest.security=TRACE
log4j.logger.org.jasig.cas=TRACE
log4j.logger.net.sf.ehcache=TRACE

# A1 is set to be a ConsoleAppender.
log4j.appender.A1=org.apache.log4j.ConsoleAppender

# A1 uses PatternLayout
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d %-4r [%t] %-5p %c %x - %m%n

log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=target/rest-api.log
log4j.appender.R.MaxFileSize=100MB
log4j.appender.R.MaxBackupIndex=3
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=%p %t %c - %m%n
```

In *rest-api-runtime.properties* file, please enable message logging option as below.

```
# Determines whether or not to enable the REST request and response message logging on the server side.
# To enable the message logging, set this property to TRUE, and enable DEBUG logging level for the package 'com.emc.documentum.rest.log' in log4j.
# The default value is false.
rest.message.logging.enabled=true

# Specifies the logging buffer size in byte for requests and responses when the message logging is enabled.
# The value MUST be a non-negative integer.
# The default value is 1048567.
rest.message.logging.buffer=10240
```

The log file will be found at the location as specified by *log4j.appender.R.File* in *log4j.properties*.

## CAS Server Logging

CAS server logging is configured by the cas.war\WEB-INF\classes\log4j.properties file. Please enable debugging level logging for the following Java packages.

```
<logger name="org.springframework.webflow" additivity="true">
   <level value="DEBUG" />
   <appender-ref ref="cas" />
</logger>
<logger name="org.jasig" additivity="true">
   <level value=" DEBUG" />
   <appender-ref ref="cas" />
</logger>
<logger name="net.sf.ehcache" additivity="true">
   <level value="DEBUG" />
   <appender-ref ref="cas" />
</logger>
```

The log file can be found at the location specified by the CAS appender.

```
<appender name="cas" class="org.apache.log4j.RollingFileAppender">
    <param name="File" value="cas.log" />
    <param name="MaxFileSize" value="512KB" />
    <param name="MaxBackupIndex" value="3" />
    <layout class="org.apache.log4j.PatternLayout">
       <param name="ConversionPattern" value="%d %p [%c] - %m%n"/>
    </layout>
 </appender>
```

It is also helpful to capture the HTTP access logs for the web container. For instance for Tomcat server, it can be configured in *<tomcat>\conf\server.xml*.

```
<Host>
  <Valve className="org.apache.catalina.valves.AccessLogValve" directory="logs"
           prefix="localhost_access_log." suffix=".txt"
           pattern="%h %l %u %t &amp;quot;%r&amp;quot; %s %b" />
</Host>
```

Then HTTP access logs can be found in *<tomcat>\logs\localhost_access_log.txt*:

```
192.168.0.6 - - [19/Dec/2013:18:45:18 -0800] "POST /cas/login?service=https%3A%2F%2F
rstomcat2%2Fdctm-rest%2Frepositories%2Facme01 HTTP/1.1" 302 -
192.168.0.5 - - [19/Dec/2013:18:45:20 -0800] "GET
/cas/proxyValidate?pgtUrl=https%3A%2F%2Frstomcat2%3A8443%2Fdctm-
rest%2Fcas%2Fproxy%2Freceptor&ticket=ST-1-V0UdqjHbccqgevAfPmFl-RSCAS02.EMC.COM
&service=https%3A%2F%2Frstomcat2%2Fdctm-rest%2Frepositories%2Facme01 HTTP/1.1" 200 299
192.168.0.5 - - [19/Dec/2013:18:45:20 -0800] "GET /cas/proxy?pgt=TGT-2-
aD5Kg2ZIzEjr9RRrxC5JaXf5rYFG0bYVtx7bPI4vwshttyEYZ6-
RSCAS02.EMC.COM&targetService=ContentServer HTTP/1.1" 200 207
192.168.0.10 - - [19/Dec/2013:18:45:27 -0800] "GET
/cas/proxyValidate?service=ContentServer&ticket=ST-2-sAFAvL1WSIZ1BgJgYNFB-RSCAS02.EMC.COM
HTTP/1.1" 200 431
```

On the CAS server, it is strongly recommended dumping all the CAS configuration files for thorough analyze, including:

- cas.war/WEB-INF/cas.properties
- cas.war/WEB-INF/web.xml
- cas.war/WEB-INF/deployerConfigContext.xml
- cas.war/WEB-INF/view/jsp/protocol/2.0/casServiceValidaitonSuccess.jsp
- cas.war/WEB-INF/spring-configuration/ticketRegistry.xml
- cas.war/WEB-INF/classes/ehcache-replicated.xml

### REST Server Troubleshooting

Here are general steps to verify the deployment of REST server:

- Review all settings in the *rest-api-runtime.properties* file.
- SSL is required to be setup for the Web Container.

  Please refer to container documentation for the SSL setting up. If the error message contains '*PKIX path building failed*', it is usually caused by invalid SSL setting.

- Enable INFO/DEBUG/TRACE logging level.
- Check the server startup log.

  The following log will be seen for the successful CAS SSO start: "*Authentication mode is set to ct-cas for the Documentum Core REST Services.*"

- Verify home document resource.

  For instance, *https://rest:8443/dctm-rest/services* should return the resource representation of home document (anonymous access).

Please make sure the Content Server connectivity is good.

- Review all settings in the *dfc.properties* file.
- Verify repositories resource is available.

  For instance, *https://rest:8443/dctm-rest/repositories* should return the repository feed (anonymous access).

Please also make sure the CAS server connectivity is good.

- Verify the CAS server can be reached from the REST server.

  For instance, try *https://cas:8443/cas/serviceValidate?ticket=Faked&service=http://localhost* .It is expected to get an error XML page.

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
        <cas:authenticationFailure code='INVALID_TICKET'>
                ticket &#039;Faked&#039; not recognized
        </cas:authenticationFailure>
</cas:serviceResponse>
```

### Content Server Troubleshooting

First of all, it's necessary to verify that the CAS SSO plugin for Content Server has been enabled and loaded successfully. For any failure related to the plugin, please review all settings in *dm_cas_auth.ini* and tracing logs in *‹DOCUMENTUM›\dba\log*.

Then please verify that the LDAP server configuration for each repository has been setup correctly. For any user authentication failure, please review the LDAP configuration using Documentum Administrator, and check whether LDAP users are synchronized to the repository with the right user source and the right distinguished name attribute.

Thirdly, please verify that the CAS server can be reached using a web browser. For instance, enter *https://cas:8443/cas/proxyValidate?ticket=Faked&service=ContentServer* and it is expected to get the error XML page.

For more information about Content Server troubleshooting, please refer to the white paper "*Documentum Content Server Central Authentication Service (CAS) SSO*".

### CAS Server Troubleshooting

Here is a checklist to verify the CAS server deployment.

- Verify CAS deployment:
    - As CAS server requires extensions to support the SSO, the following CAS plugin jars are required to be in ‹cas-3.5.2›\WEB-INF\lib:
        - cas-server-integration-ehcache-3.5.2.jar
        - cas-server-integration-restlet-3.5.2.jar
        - cas-server-support-generic-3.5.2.jar
        - cas-server-support-ldap-3.5.2.jar
    - SSL is required to be setup for the Web Container.

      Please refer to container documentation for the SSL setting up.
    - Please check server logs after the CAS startup completes.
    - 
    - Try CAS home login page by visiting *https://cas:8443/cas/login*
- Verify service registry for REST and CS:
    - Attribute '*dmCSLdapUserDN*' must be mapped from LDAP distinguished name in the bean of '*attributeRepository*'
    - Attribute '*dmCSLdapUserDN*' must be allowed for releasing in Content Server service registry.
    - Content Server service must be registered to support proxy.
- Verify LDAP connectivity:
    - Below beans are required to be set correctly in *‹cas-3.5.2›\WEB-INF\deployContext.xml*:
        - *bindLdapAuthenticationHandler*
        - *credentialsToLDAPAttributePrincipalResolver*
        - *ldapContextSource*
        - *attributeRepository*

- Try login to CAS with the LDAP username and password, e.g. *https://cas:8443/cas/login*

  For any failure, please check CAS server logs.

- Verify REST server connectivity:

  - Please use a web browser to verify that the REST server can be reached from the CAS Server.

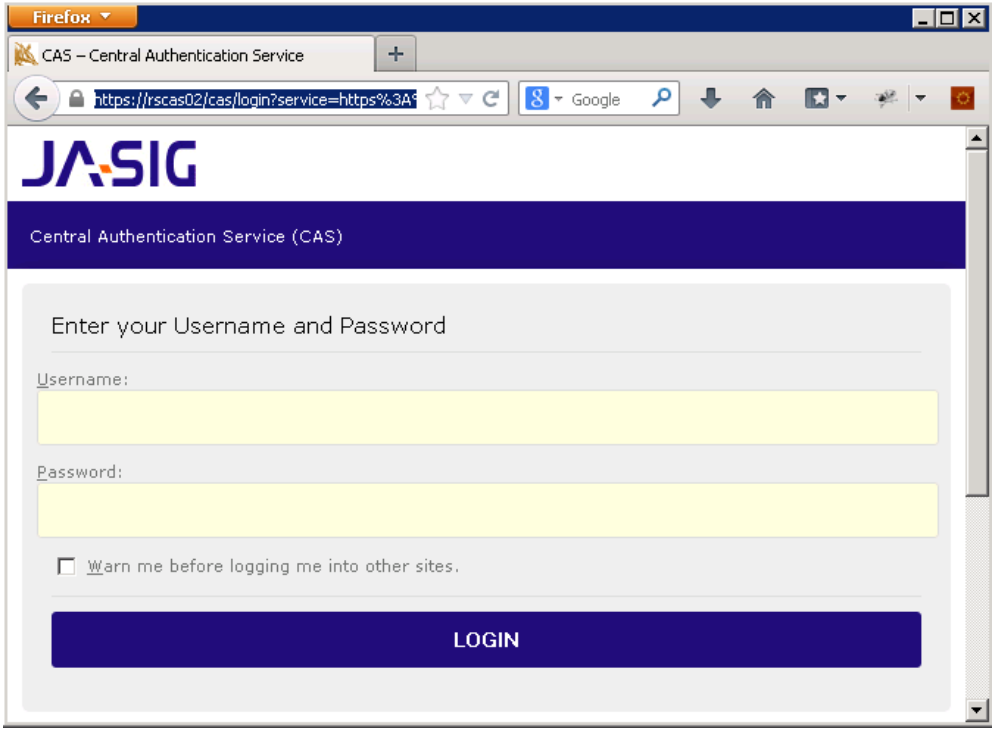
    For instance, enter this URL in a browser: *https://rest:8443/dctm-rest/cas/proxy/receptor* and a blank page will be returned (not an error page).

### Client Side Troubleshooting

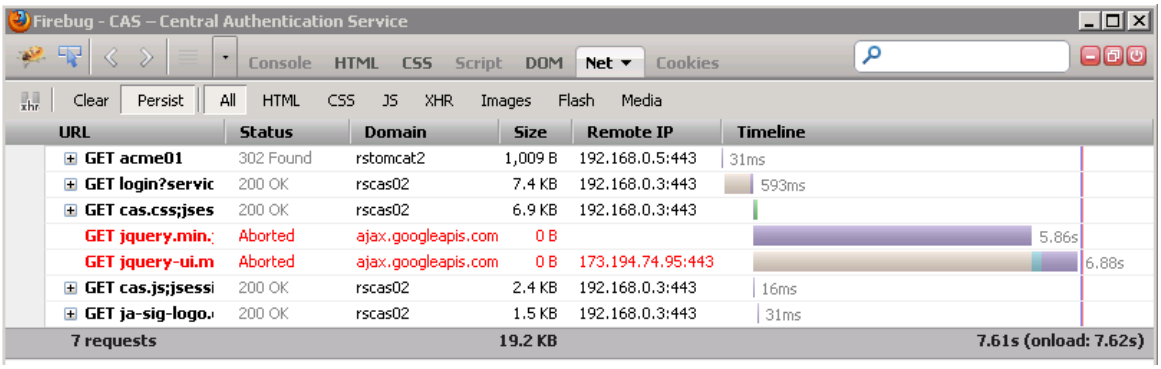
On the client side, it's useful to use a web browser to test the CAS SSO. Almost all latest versions of web browsers have the inspector plugins (or development mode). It's easy to use such kind of plugins to test the CAS SSO workflow on the client side. Here is the snapshot for Firefox 22.0 test using the Firebug add-on (http://getfirebug.com/).

**Step 1):** Redirect from resource URL to CAS login URL

Enter https://rest-server/dctm-rest/repositories/acme01 in the browser address bar, and your browser is redirected to the URL of *https://cas-server/cas/login?service=https%3A%2F%2Frest-server%2Fdctm-rest%2Frepositories%2Facme01*.

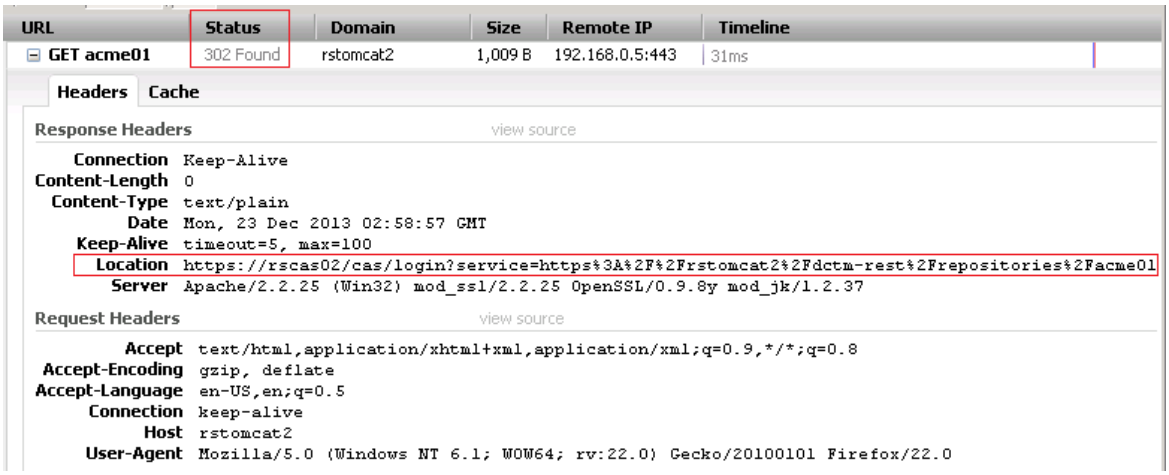



At the back-end, the activities from Firebug show the detail of the HTTP layer interactions.

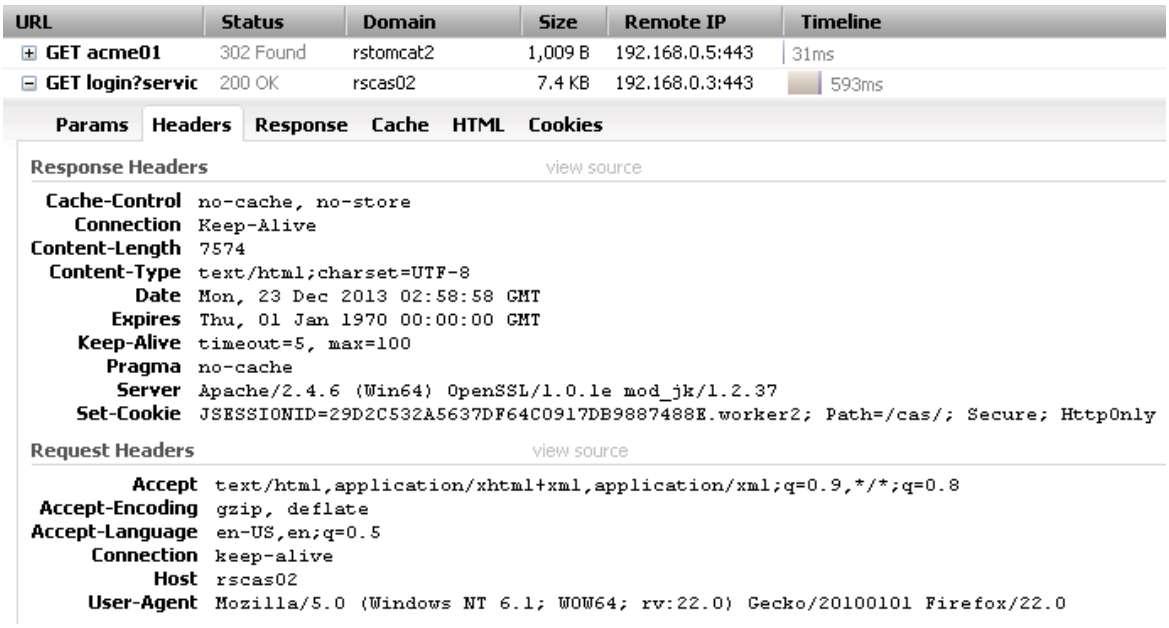

Firebug - CAS – Central Authentication Service
Console HTML CSS Script DOM Net Cookies
Clear Persist All HTML CSS JS XHR Images Flash Media
URL Status Domain Size Remote IP Timeline
GET acme01 302 Found rstomcat2 1,009 B 192.168.0.5:443 31ms
GET login?servic 200 OK rscas02 7.4 KB 192.168.0.3:443 593ms
GET cas.css;jses 200 OK rscas02 6.9 KB 192.168.0.3:443
GET jquery.min. Aborted ajax.googleapis.com 0 B 5.86s
GET jquery-ui.m Aborted ajax.googleapis.com 0 B 173.194.74.95:443 6.88s
GET cas.js;jsessi 200 OK rscas02 2.4 KB 192.168.0.3:443 16ms
GET ja-sig-logo. 200 OK rscas02 1.5 KB 192.168.0.3:443 31ms
7 requests 19.2 KB 7.61s (onload: 7.62s)

And for each HTTP access, there is detailed information.

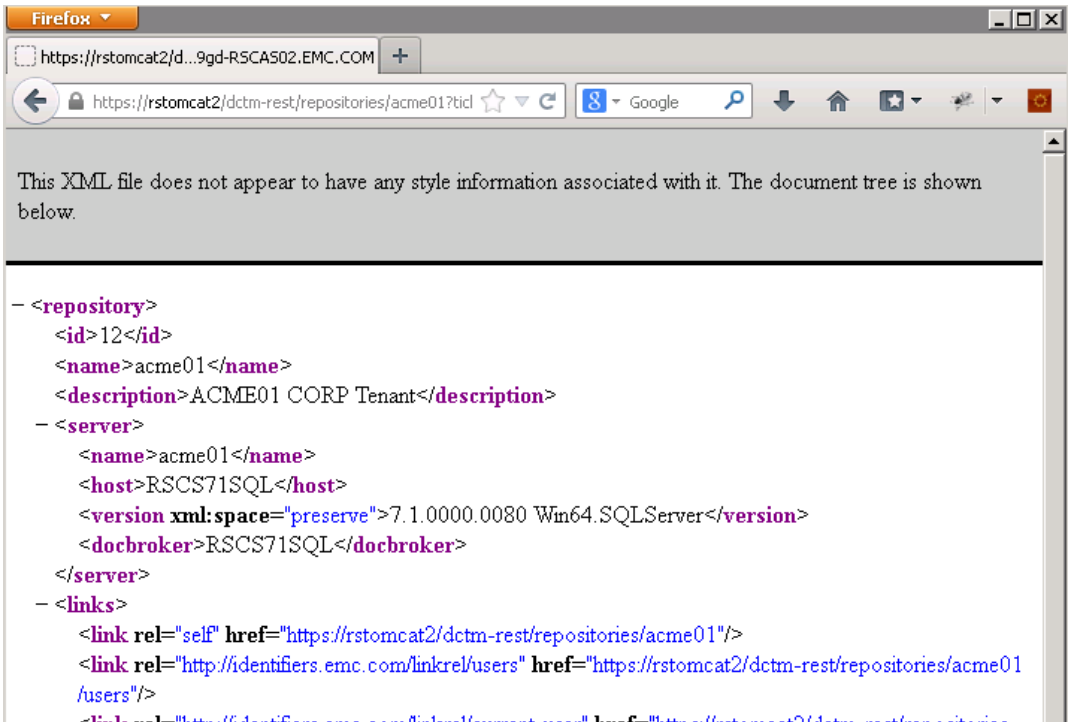
Get the repository resource URL:




Navigate to the CAS login page:



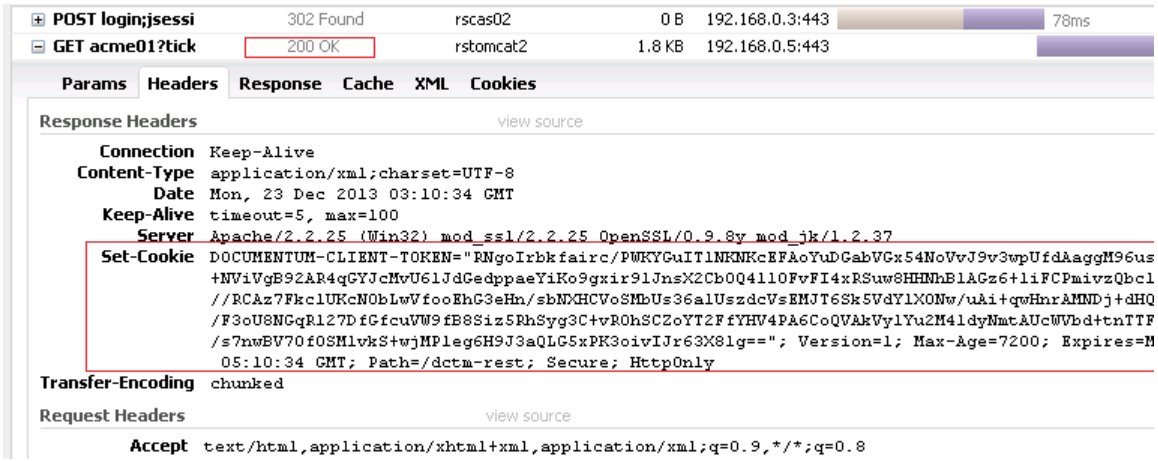
**Step 2):** Then enter the LDAP user name and password. Finally, the repository resource is returned.



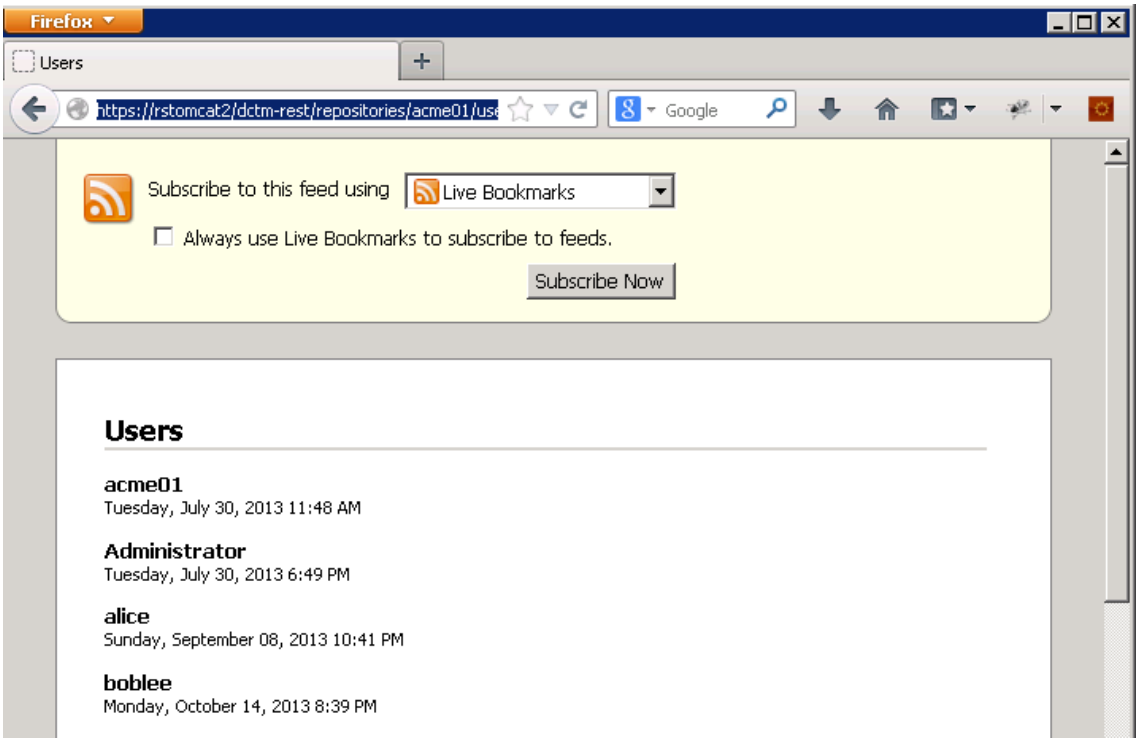

And at the back-end, further activities from Firebug are tracked.

If we look at the details, the first request posts the user credentials to the CAS server, and the CAS server redirects the web browser to the REST repository resource in its response. In addition, a CASTGC cookie is sent back.

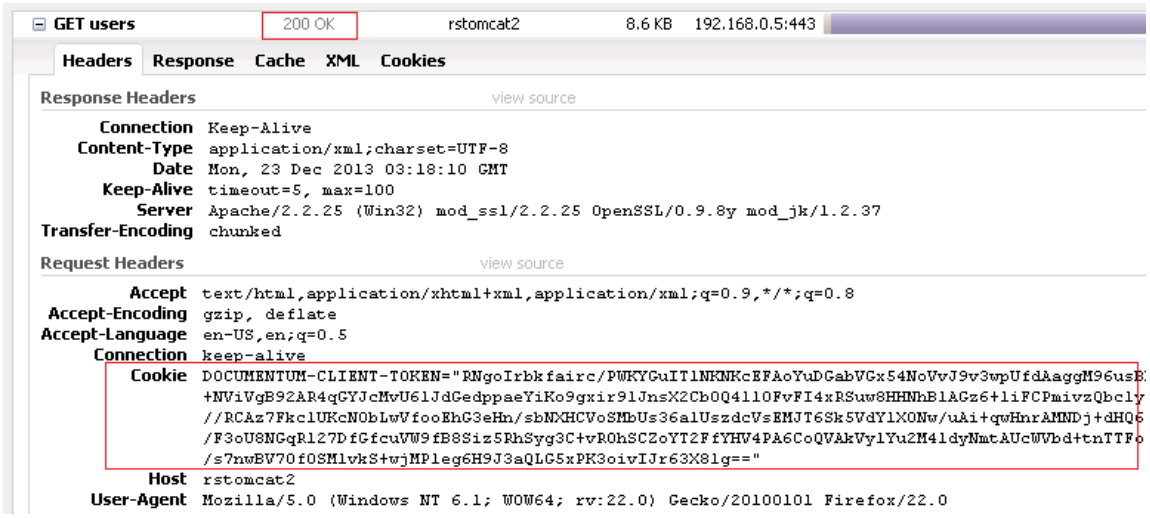
The next request reaches the REST server. In the resource response, a new DOCUMENTUM-CLIENT-TOKEN cookie is set back to the web browser.




If another resource is accessed immediately after that in the same browser, the resource is accessible without entering credentials again. For instance, access *https://rest-server/dctm-rest/repositories/acme01/users*.




And from Firebug, you can find that the browser is using the DOCUMENTUM-CLIENT-TOKEN to authenticate the user.




These snapshots demonstrate an intuitive way to trouble the CAS SSO on the client side.

## Conclusion

This paper explains the architecture of CAS, and how CAS SSO can be enabled in Documentum REST Services. For further information on CAS protocol and configuration, please refer to the Jasig CAS project site. For feature requests and successful stories on the Documentum REST Services, please contract EMC product manager for Documentum REST Services. For further information on the CAS SSO integration for Documentum REST Services, please contact EMC support.

## References


- EMC Support: http://support.emc.com
    - Documentum Platform REST Services 7.1 Development Guide
    - Documentum Platform REST Services 7.1 Release Notes
    - Documentum Content Server 7.1 Administration and Configuration Guide
    - DOCUMENTUM CONTENT SERVER CENTRAL AUTHENTICATION SERVICE (CAS) SSO A Detailed Review
- CAS project site: http://www.jasig.org/cas
- CAS User Manual Wiki: https://wiki.jasig.org/display/CASUM/Home
- CAS RESTful API: https://wiki.jasig.org/display/CASUM/RESTful+API
- CAS LDAP: https://wiki.jasig.org/display/CASUM/LDAP
- CAS Services Management: https://wiki.jasig.org/display/CASUM/Services+Management
- CAS Cluster: https://wiki.jasig.org/display/CASUM/Clustering+CAS
- Ehcache Replication: http://ehcache.org/documentation/replication